Univerza v Ljubljani

Fakulteta za računalništvo in informatiko

PORAZDELJENI SISTEMI

za modeliranje, paralelno programiranje in procesiranje

Andrej Dobnikar in Uroš Lotrič

Ljubljana, september 2008

CIP - Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

004.42(075.8)

DOBNIKAR, Andrej
Porazdeljeni sistemi za modeliranje, paralelno programiranje in procesiranje / Andrej Dobnikar in Uroš Lotrič. - 1. izd. - Ljubljana : Fakulteta za računalništvo in informatiko, 2008

ISBN 978-961-6209-63-2
1. Lotrič, Uroš
240990208

Copyright © 2008 Založba FE in FRI. All rights reserved.
Razmnoževanje (tudi fotokopiranje) dela v celoti ali po delih brez predhodnega dovoljenja Založbe FE in FRI prepovedano.

Š 004.4 DOBNIKAR, A.
Porazdeljeni...

UNIVERZA V LJUBLJANI Knjižnica FE in FRI

0 2000557 56/3.11.08

Recenzenta: doc. dr. Branko Šter, prof. dr. Borut Robič

Založila: Fakulteta za računalništvo in informatiko, 2008
Urednik: mag. Peter Šega

Natisnil: KOPIJA Mavrič, Ljubljana
Naklada: 200 izvodov
1. izdaja

Kazalo

Poglavje 1

Uvod

Računalništvo je ena najmlajših znanstvenih disciplin, saj njeni začetki segajo v sredino prejšnjega stoletja. Pa vendar je njen ustanovitelj von Neumann že v šestdesetih letih intenzivno raziskoval celične paralelne strukture, ki naj bi pohitrile procesiranje in hkrati posnemale ustroje in delovanje živih organizmov. Njegove ideje o celičnih avtomatih predstavljajo rojstvo porazdeljenih sistemov. Od tedaj pa do danes so se številni novi dosežki na področjih računalniških tehnologij, od mikroelektronske, aparaturne in programske, prenašali tudi na področje porazdeljenih sistemov. Šele z razvojem računalniških omrežij in večjedrnih računalniških sistemov pa je bila zares dana podlaga za široko uveljavitev idej o sistemih za paralelno procesiranje. Temu so sledile raziskave programskih orodij kot so knjižnice MPI in OpenMP za paralelno programiranje kot tudi platform, na primer Globus, Condor, Alchemi, za procesiranje v okolju GRID, oziroma v omrežju računalniških sistemov brez omejitev. S takšno računalniško tehnologijo je mogoče danes reševati izjemno kompleksne probleme, ki zahtevajo reševanje na osnovi različnih naravnih algoritmov. Ti delujejo po vzoru delovanja živih organizmov, njihovih razvojnih zakonitosti in simboličnem načinu sklepanja. Ker so takšni algoritmi praviloma zelo zahtevni glede procesorske moči in porabe pomnilnika, je njihova izvedba s porazdeljenimi računalniškimi tehnologijami danes edina realna možnost.

V knjigi so podane osnove porazdeljenih sistemov za modeliranje in paralelno programiranje oziroma procesiranje. Najprej so opisane značilnosti celičnih avtomatov, s katerimi je mogoče na diskreten in kvantiziran način opisovati dinamiko najrazličnejših sistemov. Opisan je način sinteze poljubnega dinamičnega sistema v strukturne značilnosti celičnega avtomata oziroma v omrežje avtomatov. Sledi opis stanja paralelnih računalniških sistemov, predvsem z vidika povezovanja univerzalnih računalniških enot in vrednotenja paralelnosti procesiranja. Osrednji del predstavlja metodologija paralelnega snovanja in opis knjižnic za paralelno programiranje MPI in OpenMP. Opis značilnosti okolja GRID in posredovanja zahtev za paralelno procesiranju je predmet naslednjega poglavja, kjer je podrobneje opisana platforma

Condor. V nadaljevanju je podana primerjava med okoljem GRID in sistemi P2P, predvsem z ozirom na lastnosti sodobnih kompleksnih računalniških omrežij, kot so Internet in svetovni splet. Ob koncu so podani še trendi razvoja porazdeljenih sistemov, ki gredo v smeri mobilega in vsenavzočega računanja ter univerzalnih grafičnih procesorjev.

V prihodnje je mogoče pričakovati še bolj intenzivno raziskovanje različnih možnosti za paralelno procesiranje in programiranje. Kot do sedaj, bo pri tem v ospredju primerjava z naravnimi modeli, ki jih zaokrožajo živi organizmi, njihovi ustroji in njihove zakonitosti procesiranja. Zato se obravnavana tematika dobro prilega področju mehkega računanja, kjer so podrobneje opisane tehnike umetnih nevronskih mrež, evolucijski algoritmi in mehka logika. Obe disciplini se bosta v prihodnje prav gotovo še bolj intenzivno prepletali kot do sedaj in s tem bogatili zakladnico znanja za nov zagon v razvoju računalniške znanosti.

Poglavje 2

Celični avtomati

2.1 Uvod

Celične avtomate je v šestdesetih letih prejšnjega stoletja vpeljal von Neumann kot orodje za preučevanje obnašanja in modeliranje kompleksnih (dinamičnih) sistemov [1, 2, 3, 4, 5, 6, 7]. Definiral jih je kot dinamične sisteme, diskretne v času in prostoru. Celični avtomat je D-dimenzionalna mreža celic, ki so končni avtomati. Sosednost celice določajo okoliške celice, ki vplivajo na prehajanje stanj celice.

V primeru $D = 2$ se sosednost petih celic (center - celica sama, sever, jug, vzhod, zahod) imenuje von Neumann-ova sosednost, sosednost devetih (vključno s sosedi na diagonalah) pa Moore-ova sosednost. Novo stanje celice α s sosednostjo $n(\alpha)$ določa enačba:

$$v^{t+1}(\alpha) = f(n^t(\alpha)) \quad . \tag{2.1}$$

Pri tem na stanje celice v času $t + 1$ vpliva celotna sosednost v času t, kar pomeni stanje celice same in stanje okoliških celic. Funkciji f pravimo tudi pravilo celičnega avtomata, ki ga določa par $(n^t(\alpha), v^{t+1}(\alpha))$ oziroma ob upoštevanju vseh kombinacij ustrezna pravilnostna tabela. Konfiguracija c je kombinacija stanj vseh celic v celičnem avtomatu. Globalna prenosna funkcija F je sprememba globalnega stanja celičnega avtomata, oziroma stanja vseh celic hkrati:

$$c^{t+1} = F(c^t) \quad , \tag{2.2}$$

kjer je $c \in C$, pri čemerj je C množica vseh konfiguracij. Celični avtomat je uniformen ali homogen, če so funkcije f vseh celic enake. Sicer je nehomogen ali neuni-

formen. Wolfram [8] je na osnovi sistematične študije obnašanja celičnih avtomatov glede na njihovo dinamiko definiral štiri kvalitativne razrede:

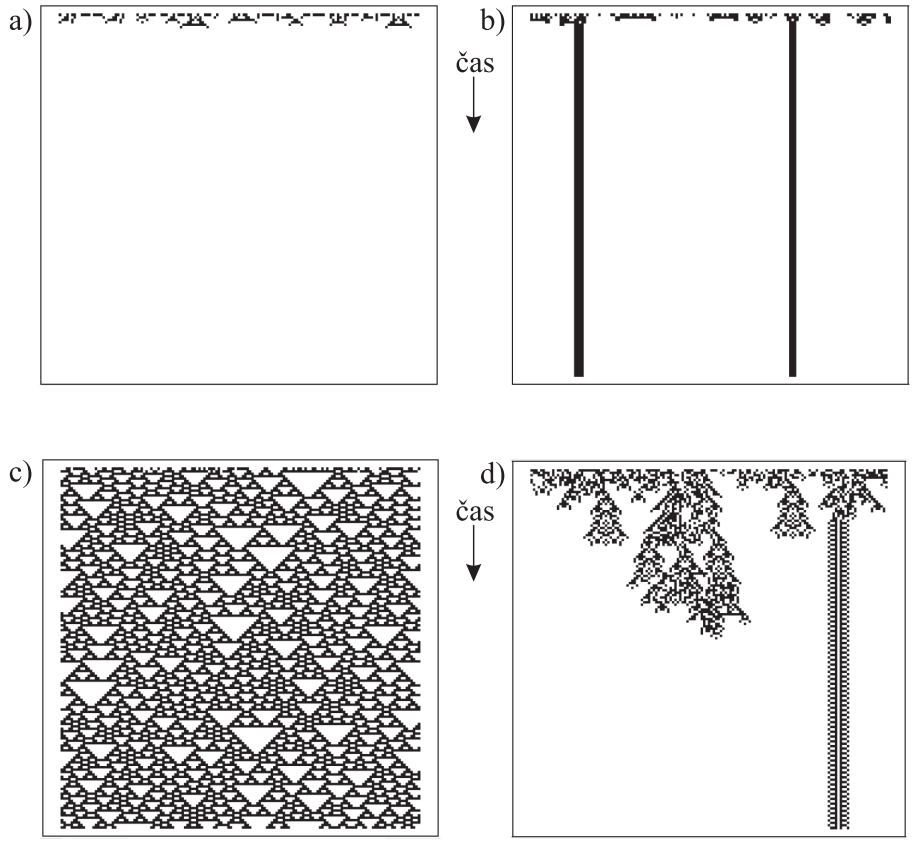
a) celični avtomati, ki prehajajo (relaksirajo) v homogeno limitno stanje,

b) celični avtomati, ki konvergirajo v preproste ločene periodične strukture (limitni cikli),

c) celični avtomati, ki izkazujejo kaotične aperiodične vzorce,

d) celični avtomati, ki generirajo kompleksne vzorce lokalnih struktur z dolgimi prehodnimi pojavi - ni primerne analogije pri zveznih dinamičnih sistemih.

Wolframovi razredi so predstavljeni na sliki 2.1.



Slika 2.1: Wolframovi razredi, predstavljeni z enodimenzionalnimi celičnimi avtomati. Horizontalno je predstavljena konfiguracija celičnega avtomata v določenem času, vertikalna os pa prikazuje konfiguracijo avtomatov v različnih časih. Vir: [10].

2.2 Univerzalno računanje s celičnimi avtomati

Univerzalna moč računanja s celičnimi avtomati ($D = 2$) je kmalu postala predmet številnih raziskav, predvsem glede na tedaj že uveljavljeni model univerzalnega Turingovega stroja. Rezultat teh raziskav je dokaz, da ne obstaja univerzalni homogeni celični avtomat z 2-stanjskimi celicami in von Neumannovo sosednostjo [9].

Iskanje univerzalnosti računanja s celičnimi avtomati je mogoče v dveh smereh: v povečanju sosednosti (ni preveč zanimivo) in v nehomogenih celičnih avtomatih.

2.2.1 Univerzalni nehomogeni celični avtomat

Univerzalni računski stroj je na primer dvo-stanjski nehomogeni celični avtomat s von Neumannovo sosednostjo. Za dokaz, da je z njim možno univerzalno računanje, je potrebno pokazati, da je s celičnimi avtomati mogoče zgraditi naslednje elemente [10]:

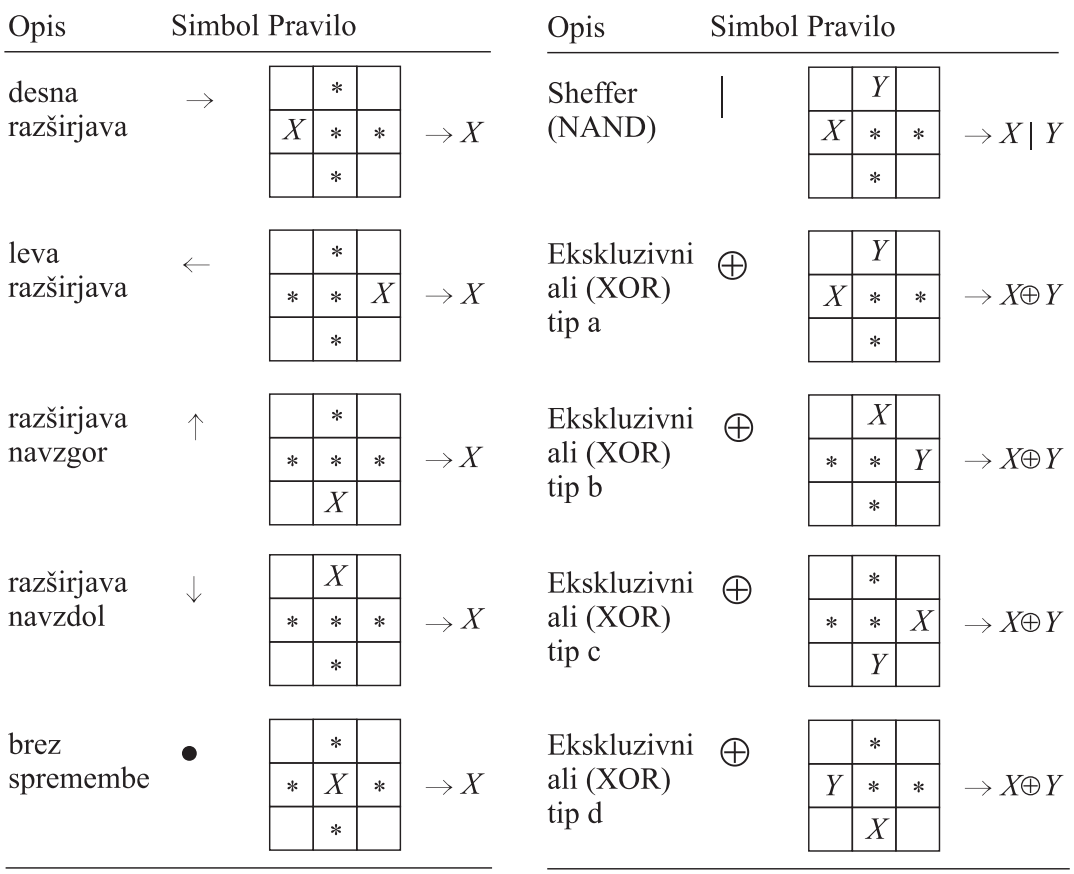
- signale in signalne poti,
- funkcijsko poln sistem logičnih operatorjev,
- sinhronizacijski pogoj (ura) oziroma verigo impulzov in
- pomnilnik.

Signali in signalne poti

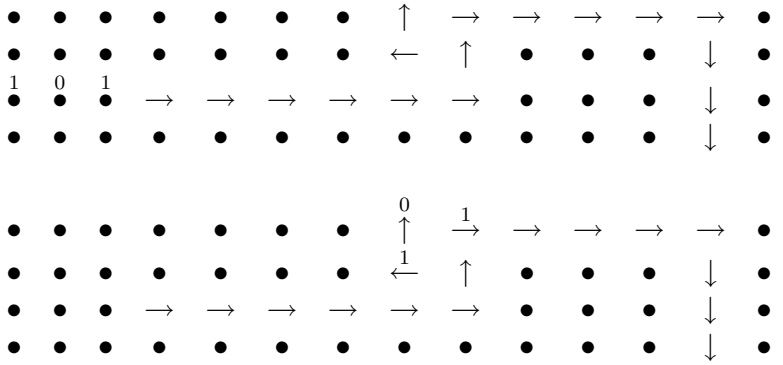
Osnovna pravila za razširjanje signalov so $\rightarrow$, $\leftarrow$, $\uparrow$ in $\downarrow$. Poleg njih so potrebni še operatorji $|$, $\oplus$ in $\bullet$. Pravila so prikazana na sliki 2.2. Vsako pravilo prikazuje vrednosti opazovane celice in njenih štirih sosed na levi ter vrednost opazovane celice po uporabi pravila na desni. Z $*$ so označena stanja celic, ki ne vplivajo na opazovano celico, X in Y pa sta binarni spremenljivki.

Operacije $\oplus$ z dvema operandoma imajo več tipov glede na lego spremenljivk X in Y. Iz tipa a dobimo z vrtenjem spremenljivk za 90^0 v smeri urinega kazalca tip b, za 180^0 tip c in za 270^0 tip d.

Primer ožičenja signalne poti v celičnem avtomatu in razširjanje signala 101 po njej je prikazano na sliki 2.3. V vsakem diskretnem časovnem koraku se signal premakne za eno celico v celičnem avtomatu. Na zgornji sliki je prikazan signal na začetku svoje poti v času $t = 0$, na spodnji pa v času $t = 10$.



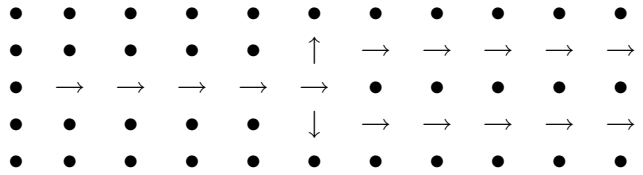
Slika 2.2: Pravila dvostanjskega univerzalnega celičnega avtomata s von Neumannovo sosednostjo.



Slika 2.3: Razširjanje signala.

Slika 2.4 prikazuje razcep signalne poti.

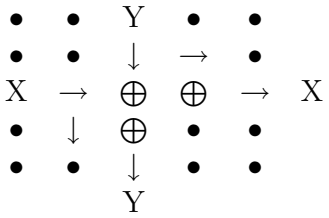
Pri brezkontaktnem prečkanju signala ne smeta vplivati eden na drugega. Možni so štirje načini prečkanja: a, b, c in d. Pri prečkanju tipa a signala prihajata iz



Slika 2.4: Razcep signalne poti

smeri sever (S) in zahod (Z) ter nadaljujeta proti jugu (J) ter vzhodu (V). Ostale kombinacije dobimo z vrtenjem signalov v smeri urinega kazalca. Za prečkanje potrebujemo najmanj 3 celice, saj moramo prenesti 2 bita v dveh smereh, kar zahteva najmanj troje logičnih vrat. Prečkanje tipa a, ki je prikazano na sliki 2.5, je mogoče izvesti s pomočjo pravil $\oplus$ tipa a z upoštevanjem zvez

$$\begin{aligned} X &= (X \oplus Y) \oplus Y \\ Y &= (X \oplus Y) \oplus X \quad . \end{aligned} \tag{2.3}$$



Slika 2.5: Prečkanje tipa a.

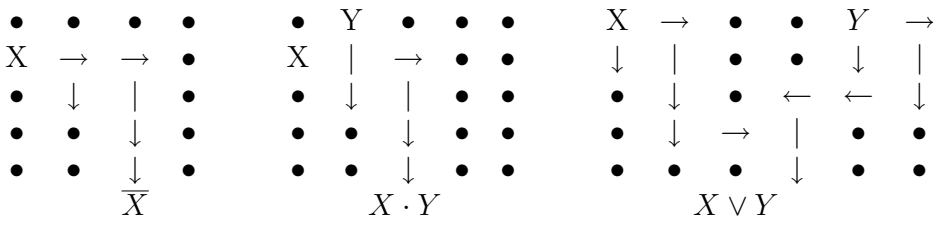
Pri homogenih celičnih avtomatih je izvedba ožičenja zelo zapletena in predstavlja praktično nerešljiv problem.

Funkcijsko poln sistem

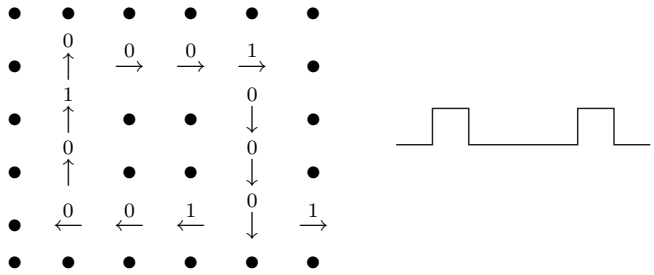
Nabor operatorjev je funkcijsko poln, če lahko z njimi izvedemo elementarne operatorje - negacijo ($^-$), in ($\cdot$) ter ($\vee$). Elementarne operatorje, izdelane s celičnimi avtomati, prikazuje slika 2.6.

Sinhronizacijski pogoj

Sinhronizacijski pogoj ali urin signal izvedemo s signalno potjo v obliki zanke. Oblika zanke ter vrednosti celic v zanki določajo obliko urinega signala. Primer je prikazan na sliki 2.7.



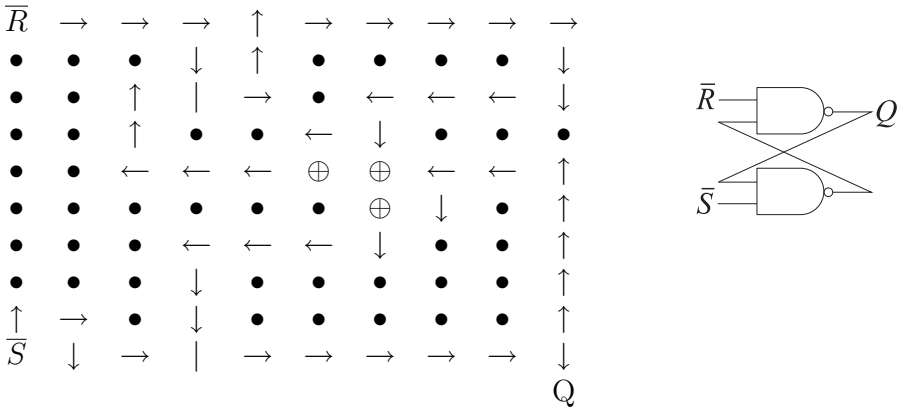
Slika 2.6: Izvedba elementarnih operatorjev s pravili $|$, $\leftarrow$, $\rightarrow$ in $\downarrow$.



Slika 2.7: Sinhronizacijski pogoj.

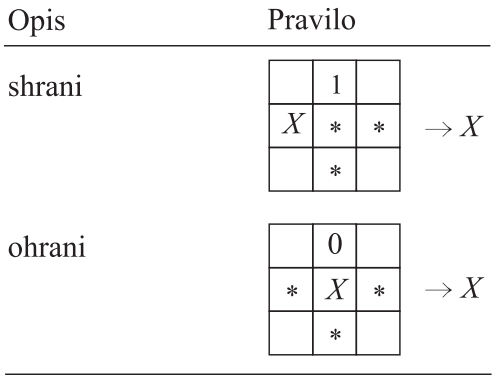
Pomnilnik

Osnova pomnilnika je pomnilna celica. Pomnilno celico RS, zgrajeno s pravili prikazanimi na sliki 2.2, prikazuje slika 2.8.



Slika 2.8: Pomnilna celica RS, izvedena s celičnim avtomatom in logičnimi vrati. Uporabljeno pravilo $\oplus$ je tipa b.

Ker je gradnja pomnilnika na zgornji način precej zapletena, je bolj smiselno definirati dve novi pravili, ki definirata eno-bitni pomnilnik. Ta glede na stanje celice na severu v centralno celico shrani vrednost celice na vzhodu ali pa ohranja stanje centralne celice (slika 2.9).



Slika 2.9: Dodatni pravili za enostavno izvedbo pomnilnika.

Število pravil je mogoče zmanjšati, če se namesto pravil za razširjanje signalov uporabimo pravilo $\oplus$ tipov a, b, c in d. Tedaj lahko vse štiri skupine elementov izvedemo le s šestimi pravili.

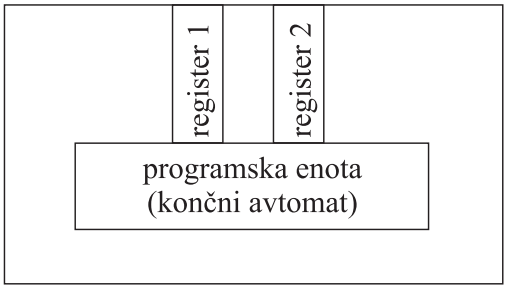
2.2.2 Izvedba univerzalnega stroja s celičnimi avtomati

Za dokaz univerzalnosti računanja lahko služi univerzalni stroj, ki ga je predlagal Minsky [11]. Sestavljen je iz

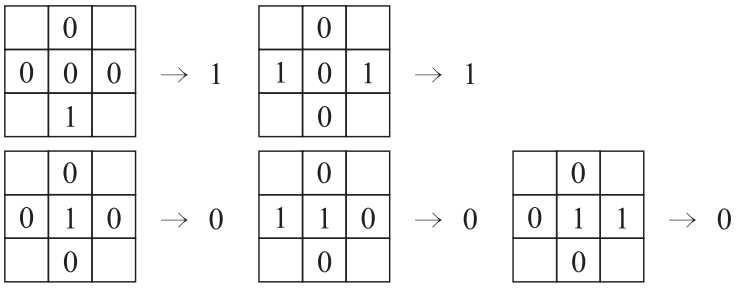
- programske enote (končnega avtomata),
- dveh registrov potencialno neskončne dolžine in
- niza naslednjih treh ukazov:
  - povečanje vsebine registra za 1,
  - zmanjševanje vsebine registra za 1 in
  - testiranje, ali je vsebina registra enaka 0.

Banks [12] je leta 1970 dokazal, da je možna izvedba univerzalnega stroja z nehomogenim celičnim avtomatom (slika 2.10). Univerzalni celični avtomat ima tri skupine pravil: v prvo skupino sodijo pravila, potrebna za izvedbo končnega avtomata, v drugo skupino sodijo pravila tipa Banks za celice dveh registrov, v ostalih področjih pa so pravila ozadja. Pravila ozadja prikazuje slika 2.11, Banksova pravila pa slika 2.12. Vse ostale kombinacije ničel in enic ohranjajo stanje centralne celice.

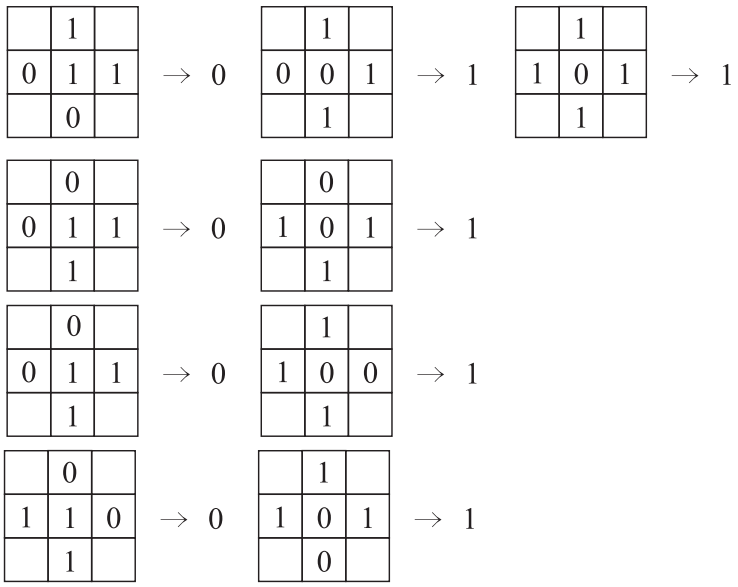
Poglejmo si nekaj primerov izvajanja ukazov z univerzalnim strojem. Ležeč pravokotnik na slikah označuje programsko enoto (PE), pokončen pa register. Število ničel v registru določa shranjeno vrednost. Z b so označene celice z vsebino 0 in pravilom ozadja, r označuje celico s stanjem 0 in pravilom B.



Slika 2.10: Univerzalni stroj Minskega, izveden s celičnim avtomatom.



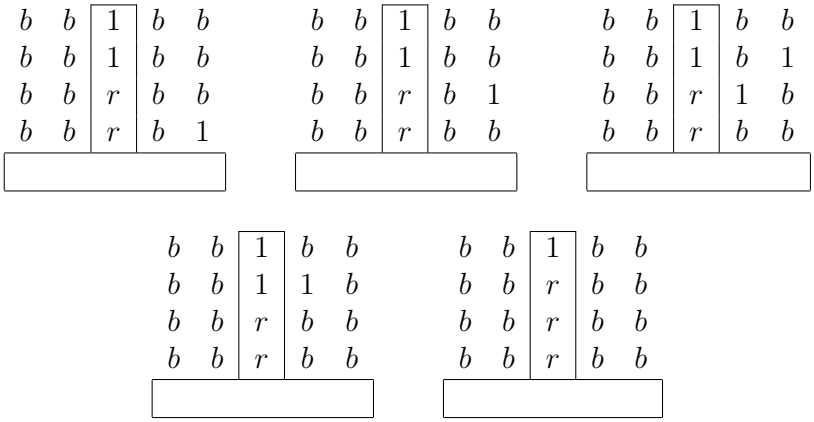
Slika 2.11: Pravila ozadja.



Slika 2.12: Banksova pravila.

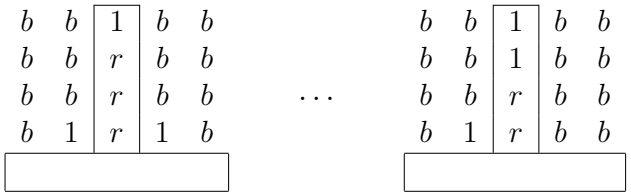
- Povečanje vsebino registra iz 2 na 3. Potek izvajanja ukaza je prikazan na sliki 2.13. Vrstica nad programsko enoto vsebuje enico, ki sproži izvedbo ukaza.

- Zmanjševanje vrednosti registra iz 3 na 2. Zmanjševanje sproži enica, ki se



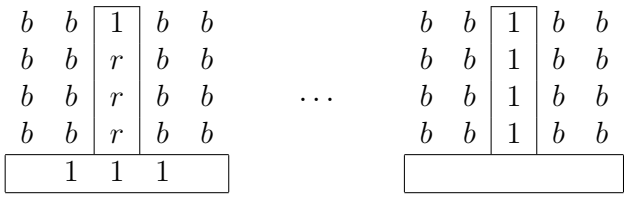
Slika 2.13: Povečevanje vsebine registra.

nahaja nad programsko enoto na desni strani registra (slika 2.14).



Slika 2.14: Zmanjševanje vsebine registra.

- Testiranje vsebine registra. Programska enota opazuje spodnjo celico registra. Če ima ta celica vrednost ena, je vrednost registra nič, sicer pa ne.
- Ničenje (*ang.* reset) vrednosti registra izvedemo s kombinacijo treh enic v programski enoti tik pod registrom (slika 2.15).



Slika 2.15: Ničenje vsebine registra.

2.3 Model ALIFE

Osnovni model celičnega avtomata je mogoče spremeniti tako, da se z njim poveča kompleksnost računanja, ter posledično omogoča reševanje zahtevnejših nalog, pred-

vsem takšnih, kot jih rešujejo živi organizmi. Tako spremenjen model se imenuje ALIFE. Model ALIFE se razlikuje od celičnega avtomata po:

- nehomogenih celicah,
- bolj kompleksnih pravilih in
- spreminjanju stanj in pravil s časom.

Celice modela ALIFE so lahko pasivne (brez pravila za spreminjanje stanj) ali aktivne (z vgrajenim pravilom). Aktivne celice lahko v enem časovnem koraku naredijo naslednje:

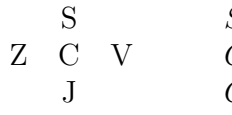
- Izvedejo dostop do lastnega stanja in stanja sosednjih celic.
- Spremenijo lastno stanje in tudi stanja sosednjih celic. Če pri tem nastopi kolizija, se le-ta rešuje naključno.
- Kopirajo lastno pravilo v sosednjo pasivno celico. Tudi tukaj se kolizija rešuje naključno.

Model ALIFE dovoljuje, da opazovana celica spreminja stanja tudi sosednjim celicam in vanje kopira nova pravila. Zaradi naključnega reševanja kolizij je model ALIFE nedeterminističen. To je bistvena značilnost živih organizmov.

2.3.1 Evolucija v prostoru pravil

Omejimo se na dvo-stanjske celice s von Neumannovo sosednostjo. Vsaka celica modela ALIFE ima genotip ali genom (tudi kromosom), ki podaja pravilo celice. V primeru petih sosednjih celic je število možnih sosednostnih kombinacij $2^5 = 32$. Vsaki sosednostni kombinaciji ali naslovu ustreza gen, ki določa spreminjanje stanj in kopiranje (slika 2.16). Vsak gen ima 10 bitov, pet za spreminjanje stanj in pet za kopiranje. Vsi geni skupaj tvorijo genotip ali kromosom. Gen tvorijo biti $(S_C S_S S_J S_V S_Z C_C C_S C_J C_V C_Z)$. Namesto označevanja genov z dvema vrstama bitov lahko gene opišemo tudi s petimi štiri-stanjskimi spremenljivkami $(Z_C Z_S Z_J Z_V Z_Z)$. Kodirna tabela je prikazana na sliki 2.16.

Primer: Vzemimo, da sosednostni kombinaciji (00101) ustreza gen $(01 + +-)$. Spreminjanje celic in pravil v celicah je potem sledeče: $S_C = 0$, $C_C = 0$, $S_S = 1$, $C_S = 0$, $S_J = 1$, $C_J = 1$, $S_V = 1$, $C_V = 1$, $S_Z = 0$, $C_Z = 1$.



S_X označuje spreminjanje stanja celice X ($0 \rightarrow 1$ ali $1 \rightarrow 0$)
$C_X = 1$ označuje kopiranje pravila v celico X
$C_X = 0$ označuje, da se pravilo ne kopira v X

S_X	C_X	Z_X
0	0	0
0	1	-
1	0	1
1	1	+

Slika 2.16: Celica modela ALIFE in označevanje bitov v genu.

Model ALIFE deluje tako, da vsaka celica glede na sosednostno kombinacijo izvede pravilo skladno z genom oziroma njegovimi 10 biti. To pomeni ustrezne spremembe stanj in kopiranje pravila. V primeru modela ALIFE za razliko od klasičnega celičnega avtomata sledi vsaki spremembi stanj oziroma kopiranju pravil še korak evolucije, s katerim se model prilagaja problemu. Korak evolucije sestavljajo genetski operatorji križanja in mutacije [8]:

- Križanje. Vsaka celica naključno izbere sosednjo celico, nato se izvede križanje z verjetnostjo p_{cross} nad genotipoma dveh celic (naključna izbira točke križanja v genomu in zamenjava delov na eni strani točke). Novi genotip zamenja starega v opazovani celici.
- Mutacija. Sledi križanju. Za vsako celico oziroma njen genotip se z verjetnostjo p_{mut} izberejo biti iz niza v genomu, ki se invertirajo.

Z modelom ALIFE so bile izvedene naslednje simulacije, ki sodijo v domeno opravil živih organizmov [10]:

- samo-reprodukcijska zanka,
- mobilnost vzorca v polju,
- rast in ponovitev ali replikacija ter
- koncept energije in njena poraba kot posledica aktivnosti organizma.

2.4 Celični programski algoritem

Celični programski algoritem (CPA) [10] je način iskanja pravil za celice celičnega avtomata s pomočjo evolucije. Od klasičnih genetskih algoritmov [8] se razlikuje

po tem, da deluje lokalno, oziroma v okviru enega celičnega avtomata (tesno sklopljen način), medtem ko genetski algoritem deluje globalno, oziroma nad populacijo celičnih avtomatov (šibko sklopljen način). Celični programski algoritem torej izvaja evolucijo na enak način, kot izvaja procesiranje - na osnovi odvisnosti od lokalnih (sosednjih) celic.

Klasični genetski algoritem je iskalni postopek, ki po zgledu iz narave, na osnovi razvoja populacije genomov (možnih rešitev), išče rešitev v dani problemski domeni. Osnovna zanka klasičnega genetskega algoritma je prikazana na sliki 2.17.

```
g := 0 // števec generacij
Inicializiraj populacijo P(g)
Oceni populacijo P(g) // funkcija prileganja
WHILE ni konca DO
    g := g+1
    Izberi P(g) iz P(g-1)
    Križanje P(g)
    Mutacije P(g)
    Oceni populacijo P(g)
END WHILE
```

Slika 2.17: Osnovna zanka klasičnega genetskega algoritma v psevdo kodi.

Celični programski algoritem združuje lastnosti osnovnega modela celičnega avtomata in nekatere lastnosti modela ALIFE [10]. V tem smislu predstavlja avtomatizirano sintezo celičnega avtomata na osnovi evolucije, ki se izvaja hkrati s procesiranjem. To pomeni, da se s pomočjo evolucije iščejo pravila prehajanja stanj (genotipi) za vse celice v okviru celičnega avtomata.

Celični programski algoritem, katerega psevdo koda je predstavljena na sliki 2.18, je v bistvu ponavljanje dveh faz.

V prvi fazi ovrednoti trenutna pravila v celicah tako, da za različne začetne konfiguracije celičnega avtomata (vsebine celic) ali globalna stanja primerja rezultate po M korakih procesiranja z želenimi vrednostmi. Ker je vseh možnih začetnih konfiguracij celičnega avtomata preveč, se v vsakem koraku postopka CPA za oceno pravil uporabi C naključnih konfiguracij. Ocenjevanje celic se izvaja s štetjem konfiguracij, pri katerih celica po M korakih preide v zahtevano stanje. M korakov procesiranja celičnega avtomata je potrebnih, da se informacija zaradi lokalnih odvisnosti dovolj dobro razširi po celotnem celičnem avtomatu. Običajno je C nekaj 100 konfiguracij, M pa je približno enak številu celic na najdaljši poti v celičnem avtomatu.

V drugi fazi na podlagi predhodnega vrednotenja celic sledi popravljanje pravil na osnovi primerjanja ocen sosednjih celic. Za popravljanje pravil se uporabljata evolucijska operatorja križanje in mutacija.

```
FOR vsako celico i v celičnem avtomatu DO IN PARALELL
    Inicializiraj tabelo pravil za celico i
    f_i = 0 // prileganje i-te celice
END PARALLEL FOR
c = 0  // števec konfiguracij
WHILE ni konca DO
    Generiraj naključno začetno konfiguracijo
    Poženi celični avtomat za M korakov
    FOR vsako celico i DO IN PARALLEL
        IF celica i deluje pravilno THEN
            f_i = f_i + 1
        END IF
    END PARALLEL FOR
    c = c + 1
    IF c mod C = 0 THEN
        FOR vsako celico i DO IN PARALLEL
            Izračunaj nf_i(c)  // število boljših sosedov
            IF nf_i(c)=0 THEN
                Ni sprememb
            ELSIF nf_i(c)=1 THEN
                Zamenjava
                Mutacija
            ELSIF nf_i(c)=2 THEN
                Zamenjava s križanjem boljših
                Mutacija
            ELSE
                Zamenjava s križanjem dveh
                naključnih in boljših sosedov
                Mutacija
            END IF
            f_i = 0
        END PARALLEL FOR
    END IF
END WHILE
```

Slika 2.18: Psevdo koda postopka CPA.

Rezultat postopka CPA je celični avtomat z nehomogenimi pravili za prehajanja stanj. Povezave oziroma sosednosti so v tem primeru enake za vse celice. V nadaljevanju bomo spoznali tudi dodatke k postopku CPA, ki vodijo tudi k rešitvam z nehomogenimi povezavami celic (omrežje avtomatov).

2.4.1 Ilustracije delovanja celičnega programskega algoritma

S celičnimi avtomati je mogoče reševati najrazličnejše probleme [4, 5, 10, 13]. Med njimi se za primerjavo metod za sintezo pravil celičnih avtomatov pogosto uporabljata problem gostote in sinhronizacije. Pri tem gre tudi za pomembno vprašanje, ali je določen problem sploh mogoče rešiti analitično oziroma klasično z deterministično programsko implementacijo in če ne, ali je evolucijski pristop primeren.

V nadaljevanju bodo podani nekateri rezultati problemov gostote in sinhronizacije s eno-dimenzionalnimi ($D = 1$) in dvo-dimenzionalnimi ($D = 2$) celičnimi avtomati, dobljenimi na tri različne načine: analitično, s klasičnim genetskim algoritmom in s postopkom CPA. Pri tem bodo uporabljene naslednje oznake:

- k - število stanj celice v celičnem avtomatu (običajno 2),
- r - radij sosednosti,
- n - število sosednih celic
- N - število celic v celičnem avtomatu,
- R - rezultat oziroma povprečje pravilnih odgovorov celic pri najmanj 104 naključnih testnih konfiguracijah.

Radij sosednosti r določa sosednost, na primer pri $D = 1$ in $r = 2$ sosednost predstavljajo celica sama, dve levi sosedi in dve desni sosedi, $n = 2r + 1 = 5$.

Zaradi enostavnejšega zapisovanja pravil se uporablja Wolframova notacija. Po njej se binarna pravila oziroma genomi podajajo v desetiškem sistemu [4, 10]. Primer je prikazan na sliki 2.19.

Kombinacija	Nova vrednost celice	
000	0	
001	0	
010	0	
011	1	
100	0	
101	1	
110	1	
111	1	$11101000_2 = 232_{10}$

Slika 2.19: Večinsko pravilo ali pravilo 232 v Wolframovi notaciji za $D = 1$, $r = 1$, $n = 3$.

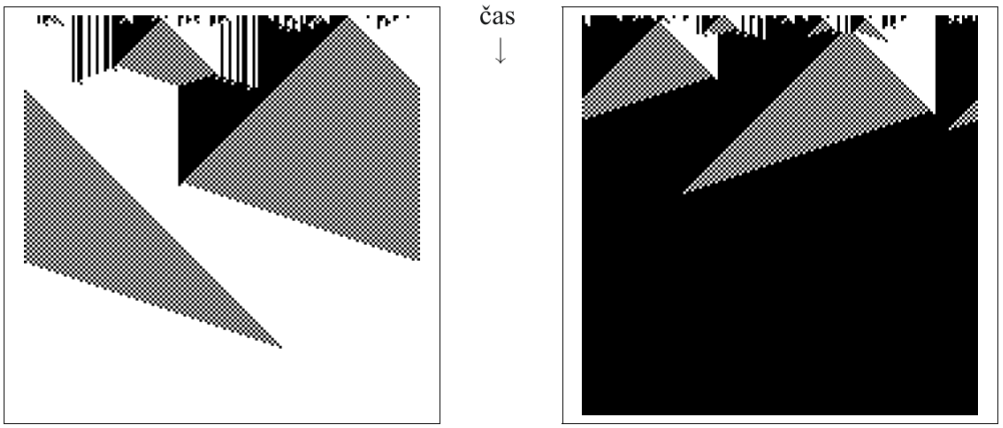
Problem gostote ($k = 2$, $r = 1, 2, 3$, $N = 149$, $D = 1$)

Problem gostote [10] predpostavlja, da bo celični avtomat znal določiti, katerih stanj je več v začetni naključni konfiguraciji. Celice v celičnem avtomatu odgovorijo pravilno, če po M korakih procesiranja pridejo v večinsko začetno stanje celic. Seveda je problem težji, kadar je število enih in drugih stanj približno enako, zato se testiranje navadno izvaja s pomočjo množice konfiguracij iz binomske porazdelitve, ki ima maksimum pri 0,5.

- Analitična rešitev. Gacs, Kurdyumov in Levin (GKL) [14] so podali naslednjo homogeno rešitev za $D = 1, r = 3$:

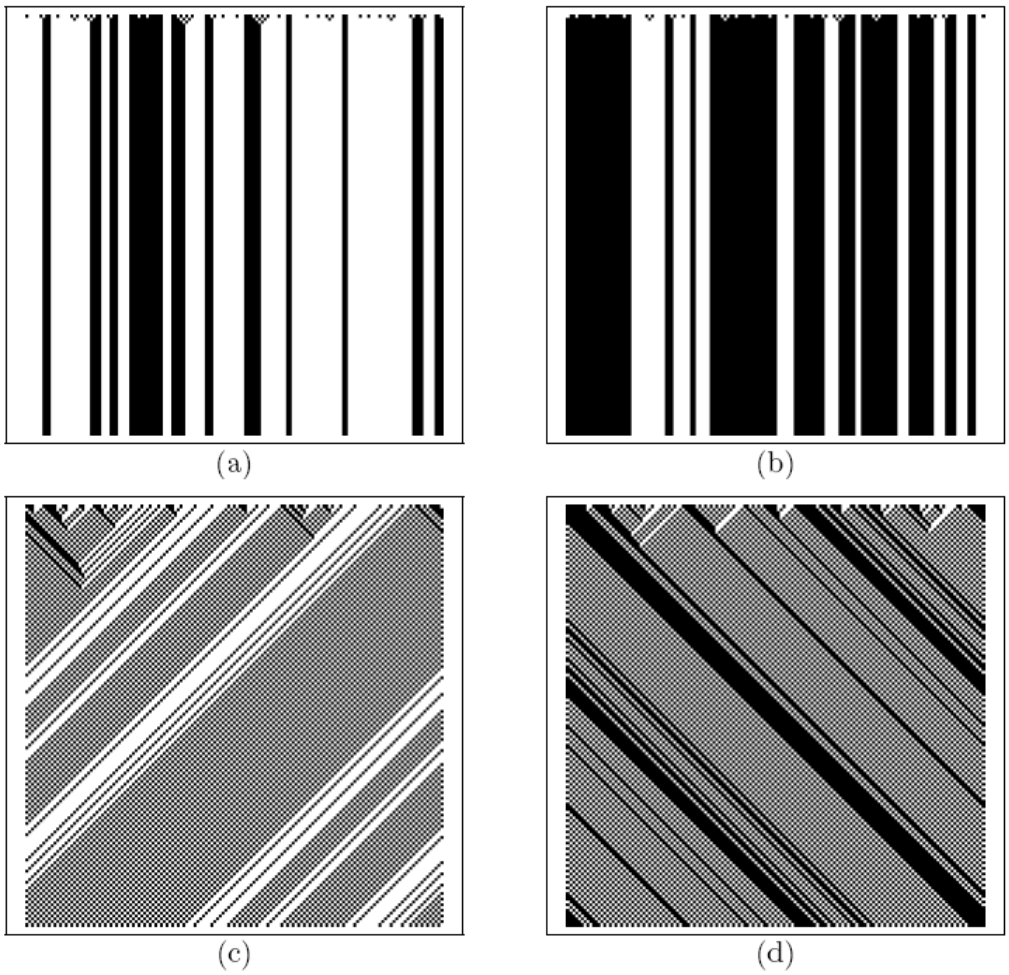
$$v_i(t+1) = \begin{cases} \text{maj}(v_i(t), v_{i-1}(t), v_{i-3}(t)) & , \text{ če } v_i(t) = 0 \\ \text{maj}(v_i(t), v_{i+1}(t), v_{i+3}(t)) & , \text{ če } v_i(t) = 1 \end{cases} , \tag{2.4}$$

  pri čemer je $\text{maj}(\cdot, \cdot, \cdot)$ večinski ali majoritetni operator. Ta rešitev da rezultat $R = 0{,}816$. Dva primera procesiranja celičnega avtomata na problemu gostote z uporabo pravila GKL sta prikazana na sliki 2.20.



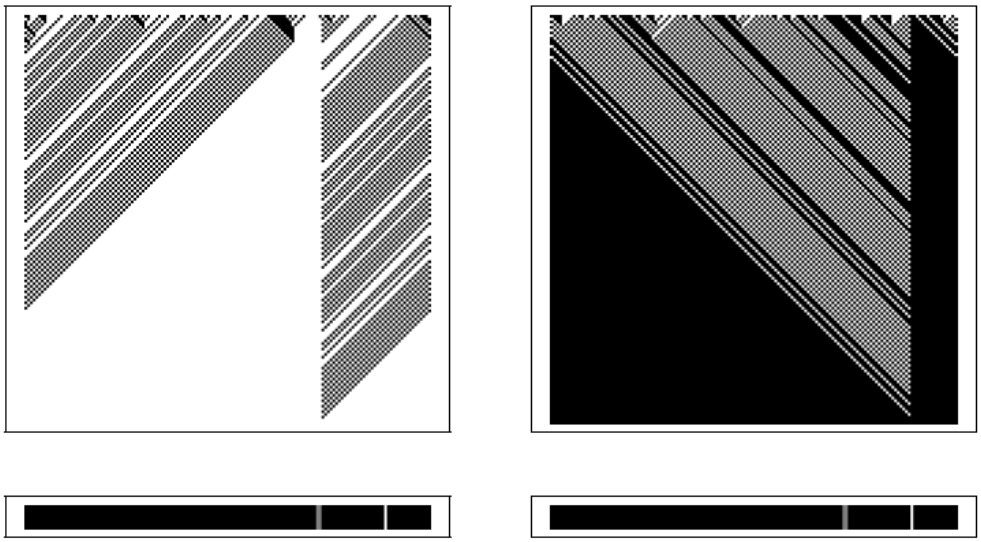
Slika 2.20: Reševanje problema gostote z uporabo pravila GKL z začetno gostoto enic enako a) 0,47 in b) 0,53. Vir: [10].

- Genetski Algoritem [8]. Z njim je mogoče iskati le homogene rešitve, saj je kombinacij sicer preveč. Za homogeni celični avtomat namreč velja, da ima $2^3 = 8$ kombinacij pri $r = 1$ in $n = 2 \cdot 1 + 1 = 3$ sosedih ter $2^8 = 256$ možnih pravil. Enako velik nehomogeni celični avtomat pa ima $(2^8)^{149} = 2^{1192} \approx 10^{400}$ možnih pravil. Z genetskim algoritmom dobimo pri $r = 1$ rezultat $R = 0{,}769$, ki ustreza homogeni rešitvi celičnega avtomata. Obnašanje homogenega celičnega avtomata z večinskim pravilom 232 in pravilom 226 na problemu gostote $D = 1,\ r = 1$ prikazuje slika 2.21.



Slika 2.21: Reševanje problema gostote z uporabo večinskega pravila 232 (a) in b)) in pravila 226 (c) in d)). Na levih slikah je začetna gostota 0,4, na desnih pa 0,6. Vir: [10].

- Celični programski algoritem. S postopkom CPA je mogoče dobiti nehomogen celični avtomat, ki z $R > 0{,}82$ presega oba zgornja rezultata. Pri tem je zanimivo, da število različnih pravil sploh ni veliko, za primer $D = 1,\ r = 1$ dobimo s postopkom CPA le tri različna pravila, ki jih uporabljajo celice znotraj celičnega avtomata. Dodatno je mogoče izboljšati rezultat, če v evoluciji spreminjamo tudi sosednost in celo število sosednjih celic. Tedaj je potrebno paziti na izbiro funkcij in njihove spremembe, saj ne smejo biti odvisne od števila spremenljivk oziroma od posameznih spremenljivk. Obnašanje celičnega avtomata z nehomogenimi pravili, dobljenimi po postopku CPA, prikazuje slika 2.22. Na spodnjem delu slike je z odtenki sivine prikazana razvrstitev pravil po celicah.

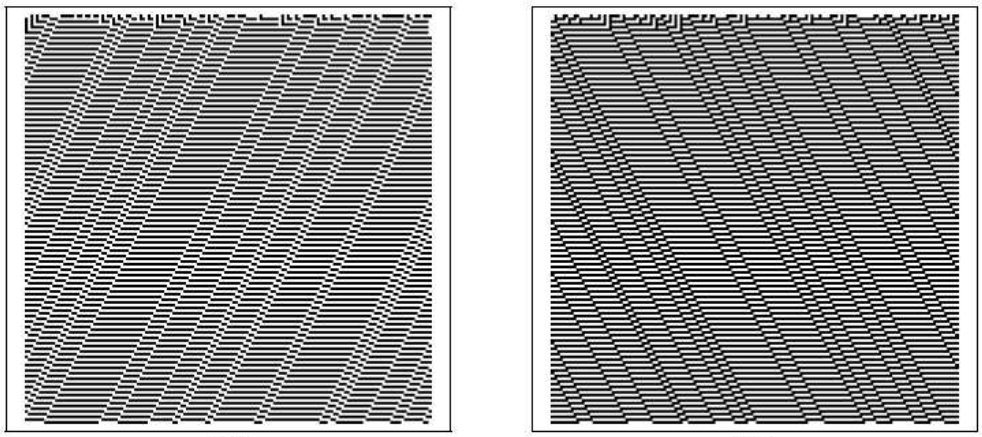


Slika 2.22: Reševanje problema gostote z uporabo postopka CPA z začetno gostoto enic enako a) 0,4 in b) 0,6. Spodaj je prikazana struktura pravil v celičnih avtomatih. Enaka pravila so označena z enakim odtenkom sive barve. Vir: [10].

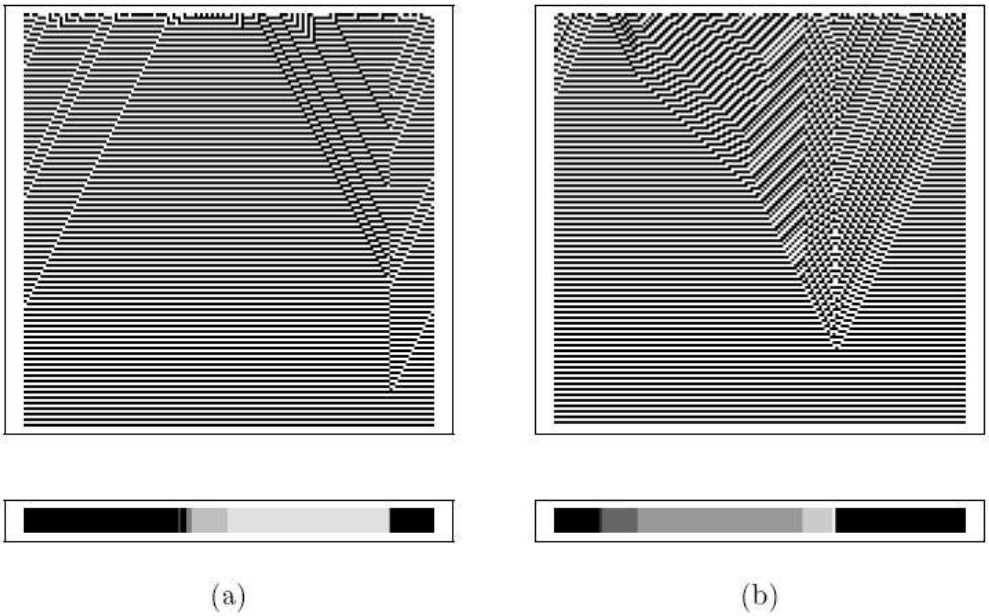
Sinhronizacija

Ideja problema sinhronizacije je, da po določenem času procesiranja vse celice v celičnem avtomatu delujejo usklajeno - vse celice hkrati spreminjajo svoje stanje iz nič v ena in iz ena v nič. Če je gostota enic po M korakih večja kot 0,5, se omenjeno zaporedje začne s samimi ničlami, sicer pa s samimi enicami. Ocena prileganja zahteva pri tem opazovanje stanja v korakih M, $M + 1$ in $M + 2$. S klasičnim genetskim algoritmom je mogoče priti do homogenega celičnega avtomata, ki za $D = 1,\ r = 1,\ N = 149$ daje rezultat $R = 0{,}84$. Razvoj celičnega avtomata z različnima homogenima praviloma je prikazan na sliki 2.23. S postopkom CPA je ta rezultat v veliki večini primerov presežen in je blizu idealnemu, $R \approx 1$. Dva primera sta prikazana na sliki 2.24.

Poznani so tudi primeri uporabe celičnih avtomatov na netrivialnih problemih, kot so urejanje - ničle na eno stran, enice na drugo stran, polnjenje dvodimenzionalnega pravokotnika, tanjšanje črt na sliki, naključni generator.



Slika 2.23: Dve neuspešni procesiranji homogenih celičnih avtomatov na problemu sinhronizacije: a) pravilo 21, b) pravilo 31. Vir: [10].



Slika 2.24: Dve uspešni procesiranji nehomogenih celičnih avtomatov, dobljenih s postopkom CPA, na problemu sinhronizacije. Spodaj je prikazana struktura pravil v celičnih avtomatih. Enaka pravila so označena z enakim odtenkom sive barve. Vir: [10].

2.5 Majhna občutljivost na napake

Večina klasičnih računalniških sistemov, posebno paralelnih, izkazuje zelo nizko stopnjo tolerance na napake. To velja tako za programsko kot strojno opremo. Pri

bodočih računalniških sistemih, ki bodo vsebovali tisoče in več procesnih elementov, je takšna nizka toleranca do napak lahko velik problem, saj bo verjetnost napak tedaj zelo visoka.

Omrežje avtomatov [4] je struktura, kjer so posamezne celice oziroma avtomati neregularno povezane med seboj. Do takšnega omrežja je mogoče priti s posebno obliko postopka CPA, kjer se v postopku evolucije spreminjajo tudi povezave celic in ne le njihove funkcije. V naravi obstajajo omrežja procesnih elementov, na primer nevronske mreže, Internet, svetovni splet, ki izkazujejo precejšnjo stopnjo tolerance tako do napak posameznih elementov kot tudi do napak na vhodu. Zanimivo je vedeti, kako robustno je omrežje avtomatov, če deluje v pogojih napačnih vhodov oziroma celic.

Prikazani bodo učinki naključnih napak na obnašanje celičnega avtomata ($D = 1$), dobljenega s postopkom CPA, nadalje obnašanja celičnega avtomata v odvisnosti od nivoja napak, oziroma področja napak, kjer celični avtomat deluje zadovoljivo. Zanimivi so tudi obseg in hitrost okrevanja po pojavu napake ter kako se napake širijo po sistemu, oziroma pod kakšnimi pogoji je mogoče motnje omejiti in preprečiti njihovo širitev po celem sistemu.

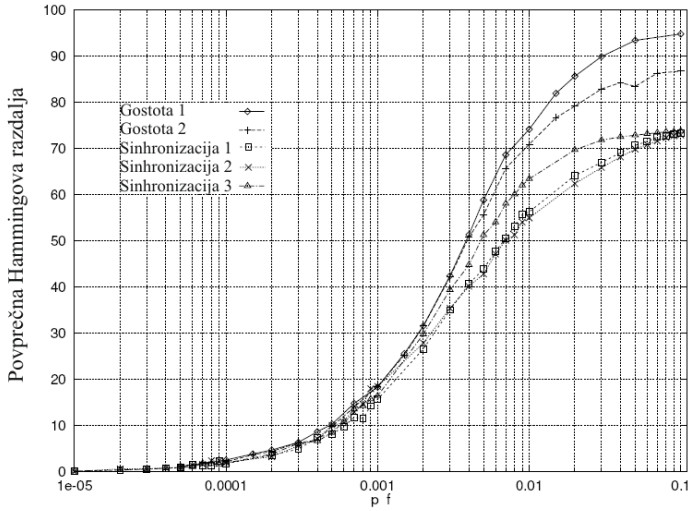
Zaradi težav pri izdelavi analitičnih modelov v zvezi z razširjanjem napak po omrežjih, je bilo največ raziskav izvedenih na simulacijah. Zelo znan je model Kauffmana [15], ki je sestavljen iz nehomogenega celičnega avtomata z neregularnimi povezavami (kar ustreza omrežju avtomatov). Pri tem modelu vsaka celica sledi pravilu prehajanja stanj, ki je naključna preklopna (Boolova) funkcija stanj sosednjih celic. Pravila in povezave so naključno izbrana na začetku, nato so v postopku procesiranja ohranjena. Tako definiran sistem omrežja avtomatov izkazuje konvergenco proti limitnim ciklom, ki jih je mogoče perturbirati z mutacijami, oziroma z naključnimi spremembami pravil. Na tej osnovi je mogoče simulirati razširjanje motenj kot funkcijo verjetnosti mutacije pravil v omrežju. Določena je bila kritična verjetnost mutacije p_{crit} nad katero se je napaka razširila po celotnem omrežju. Zelo znan je tudi Isingov model spinskih stekel, pri katerem obstaja kritična temperatura, nad katero se začetna motnja iz majhnega števila celic razširi na cel sistem.

2.5.1 Verjetnost napak v celičnem avtomatu

Opazovali bomo nehomogen celični avtomat, določen s postopkom CPA, za rešitev problema gostote in sinhronizacije. Z vpeljavo šuma pri izvajanju pravil vsake celice neodvisno z verjetnostjo p_f, postane celični avtomat nedeterminističen. Naslednje stanje celice takega avtomata se spremeni z verjetnostjo p_f glede na predvideno novo stanje iz tabele prehajanja stanj in sledi tabeli z verjetnostjo $1 - p_f$. Možne so še drugačne napake, na primer stalna okvara celice ali preklop na drugačno pravilo za nedoločen čas.

Pri simulaciji se uporabljata dva identična celična avtomata, ki delujeta vzporedno, prvi brez motenj ($p_f = 0$) in drugi z določeno verjetnostjo motenj ($p_f > 0$). Oba sistema začneta z isto začetno konfiguracijo (globalnim stanjem) v času $t = 0$, nato pa se opazuje njuno obnašanje tako, da se beleži Hammingova razdalja (število bitov pri katerih sta konfiguraciji različni). Zaradi stohastičnega delovanja se postopek večkrat ponovi, nato se rezultati statistično obdelajo (povprečijo).

Rezultati na problemih gostote in sinhronizacije kažejo, da ima Hammingova razdalja v odvisnosti od p_f obliko sigmoidne funkcije, ki je prikazana na sliki 2.25. Vsak celični avtomat je bil preizkušen za izbrano vrednost p_f nad 1000 naključnimi

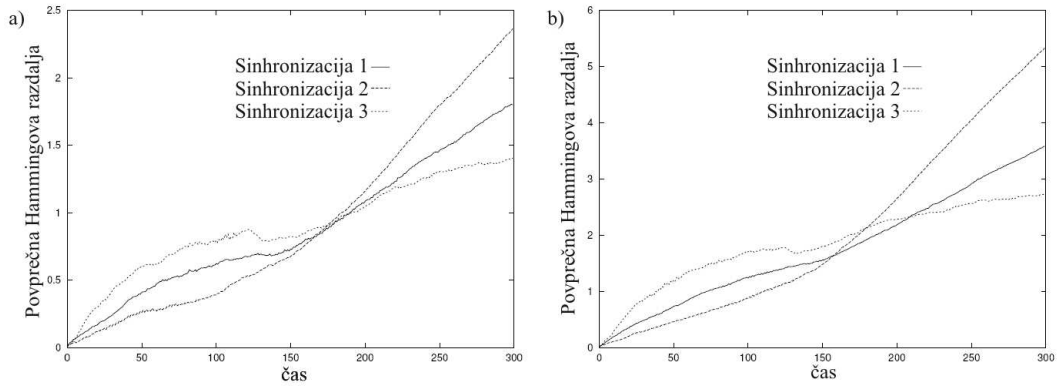


Slika 2.25: Prikaz povprečne Hammingove razdalje za pet celičnih avtomatov, ki rešujejo probleme gostote (2) in sinhronizacije (3). Število celic v celičnih avtomatih je $N = 149$. Med obema problemoma ni bistvene razlike. Vir: [10].

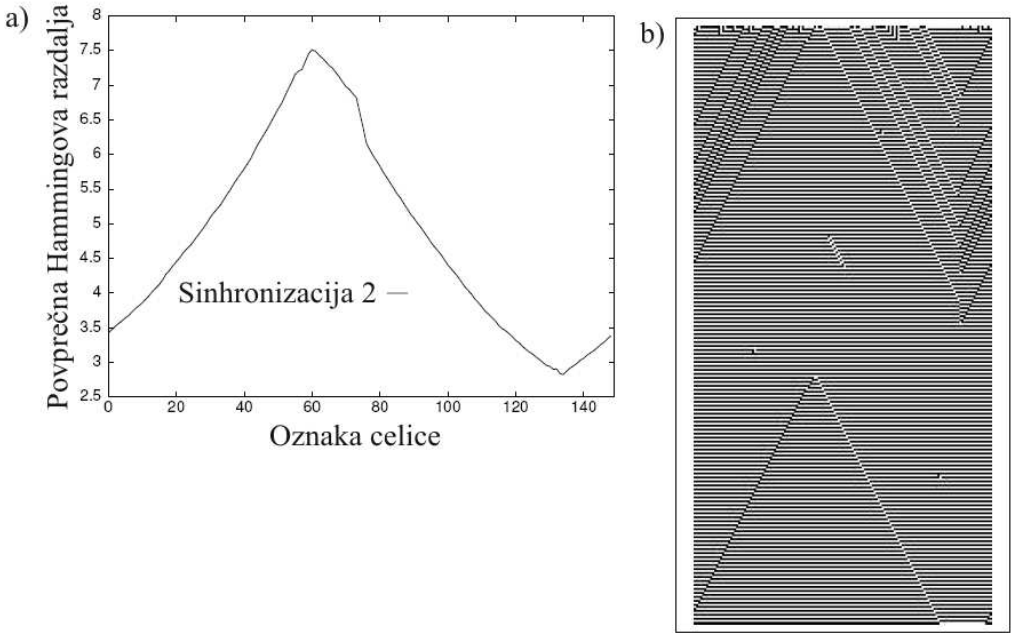
začetnimi konfiguracijami za čas 300 korakov procesiranja. Končne vrednosti celic so se nato za vsako vhodno konfiguracijo povprečile. Krivulja je pri obeh obravnavanih problemih zelo podobna. Ločimo tri področja: področje počasne rasti ($p_f < 0{,}0005$) s Hammingovo razdaljo okoli 10, področje hitre rasti ($0{,}0005 < pf \leq 0{,}01$) z razdaljo med 20 in 60 ter področje zasičenja oziroma področje zelo velike Hammingove razdalje od 70 do 100 ($p_f > 0{,}01$). Prvo področje je najpomembnejše, saj ustreza območju tolerance napak, kjer napake ne vplivajo bistveno na delovanje. To tudi pomeni, da omrežje avtomatov izkazuje zmerno degradacijo glede na napake, kar je bistveno drugače kot pri klasičnih enojedrnih računalniških sistemih, kjer je lahko že en napačen bit usoden.

Nadalje je bilo ugotovljeno, da se v območju tolerance Hammingova razdalja povečuje s časom procesiranja (slika 2.26) in da je odvisna tudi od področij različnih pravil v celičnem avtomatu, predvsem znotraj področij z enakimi pravili in na ro-

bovih področij med različnimi pravili (slika 2.27).



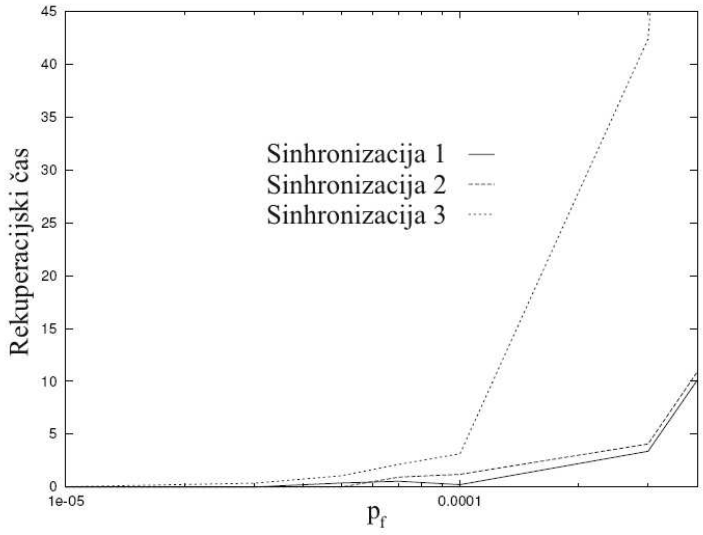
Slika 2.26: Hammingova razdalja kot funkcija časa treh celičnih avtomatov pri problemu sinhronizacije, povprečena preko 1000 naključnih konfiguracij: a) p_f = 0,00005, b) $p_f = 0{,}0001$. Vir: [10].



Slika 2.27: Problem sinhronizacije, $N = 149$: a) Vpliv lokacije pravil na Hammingovo razdaljo b) Procesiranje po motnji v 200 koraku 60-te celice. Vir: [10].

Zanimive so tudi ugotovitve glede izboljšanja Hammingove razdalje s časom. Izkaže se, da se z večjo verjetnostjo motnje p_f povečuje rekuperacijski čas, ki je definiran kot časovni interval, v katerem se Hammingova razdalja zmanjšuje za čas vsaj treh diskretnih korakov delovanja celičnega avtomata. Pri manjših verjetnostih motnje p_f je ta čas majhen, pri večjih pa naraste in to bolj pri takšnih celičnih avtomatih, pri katerih s časom najmanj narašča Hammingova razdalja (slika 2.28). Slednje

kaže na povezavo med časovnimi in prostorskimi faktorji, ki vplivajo na delovanje nehomogenih celičnih avtomatov, dobljenih z evolucijo.



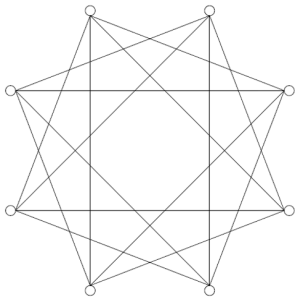
Slika 2.28: Rekuperacijski čas v odvisnosti od verjetnosti motenj p_f za tri celične avtomate na problemu sinhronizacije (N=149). Vir: [10].

2.6 Evolucija omrežnih avtomatov

V tem poglavju nas bo zanimalo, ali je mogoče izboljšati rezultate celičnega avtomata, če pri tem dopuščamo, da ima vsaka celica (avtomat) lahko različno funkcijo prehajanja stanj in različno sosednost (vplivne celice, ki pa ni nujno, da so sosedne). Tedaj govorimo o omrežju avtomatov. Izsledki so zanimivi tudi zato, ker lahko nakazujejo reševanje problema povezav med porazdeljenimi procesorji s ciljem boljših rezultatov procesiranja. Tukaj bo uporabljen izraz arhitektura za opis povezav celic celičnega avtomata oziroma omrežja avtomatov. Pojem sosednosti bo zato pomenil direktno povezane celice, ki niso nujno tudi fizično sosedne. Uporabljen bo postopek CPA za iskanje nehomogene (funkcijske) rešitve pri nestandardni arhitekturi, enaki za vse celice, in prilagojen postopek CPA za iskanje nehomogene rešitve v funkcijskem in povezovalnem smislu. V prvem primeru bo govora o fiksni arhitekturi, v drugem pa o razvijajoči arhitekturi.

Če želimo realizirati visoko zmogljivo procesiranje globalnega problema z omrežjem procesnih elementov, moramo zagotoviti visoko prepustnost informacij skozi omrežje, kar je obratno sorazmerno s povprečno razdaljo med elementi v omrežju. Pri tem je razdalja med dvema celicama enaka najmanjšemu številu povezav (vozlišč) na poti med njima. S $C_N(a, b)$ označujemo arhitekturo oziroma graf z N vozlišči

$v_i, i = 1, \ldots, N$, ki so povezani z vozlišči $v_{i\pm a}$ in $v_{i\pm b}$, obakrat po modulu N. Primer grafa za arhitekturo $C_8(2, 3)$ je prikazan na sliki 2.29.



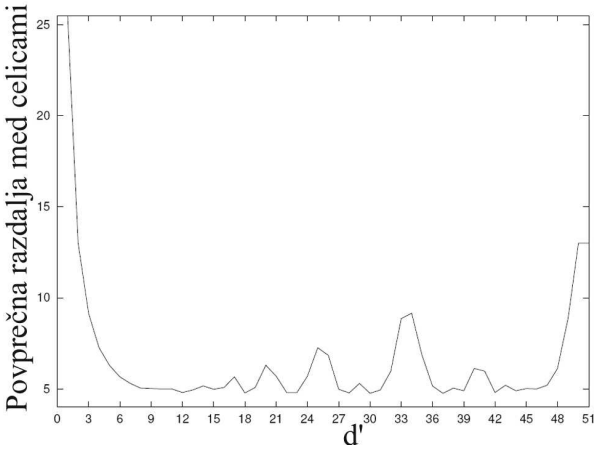
Slika 2.29: Krožni graf za arhitekturo $C_8(2, 3)$

S takšno arhitekturo lahko opisujemo sosednosti v dvo-dimenzionalnem celičnem avtomatu, ki ga razvijemo v eno-dimenzionalnega tako, da konec prejšnje vrstice povežemo z začetkom naslednje. Von Neumannova sosednost zajema sedaj dva soseda na levi in dva na desni strani vsake celice, ob predpostavki ciklične povezave celičnega avtomata. Parameter a določa bližnja soseda na levi in desni ($\pm a$), parameter b pa bolj oddaljena ($\pm b$).

Povprečna razdalja je v tem primeru zelo odvisna od a in b. Velja, da je vsaka arhitektura $C_N(a, b)$ izomorfna z arhitekturo $C_N(1, d')$, kjer je d' ekvivalent, če in samo če vsakemu paru s povezavo dolžine a (b) v $C_N(a, b)$ ustreza povezava dolžine 1 (d') v $C_N(1, d')$. Ekvivalentni parameter d' določimo tako, da najprej zapišemo v vrsto vsa vozlišča z oznakami od 0 do $N - 1$. Nato preuredimo vrsto (seznam) tako, da so vozlišča, originalno narazen za a enot, sedaj sosedna. V primeru, da je največji skupni delitelj med a in N, $\text{nsd}(a, N) > 1$, tedaj damo skupaj vozlišča, originalno narazen za b enot. Ekvivalent d' je sedaj minimalno število povezav od opazovanega vozlišča do vozlišča b (ali do a, če $\text{nsd}(a, N) > 1$). Na primer, $C_7(2, 3)$ se preslika v $C_7(1, 2)$. To je pomembno predvsem iz praktičnih razlogov, ker je prostor arhitekture $C_N(a, b)$ zelo velik.

Če želimo minimizirati ceno povezovanja, definirano z $a + b$, kot tudi maksimizirati obnašanje omrežnega avtomata, potem moramo uporabljati ustreznejšo arhitekturo, običajno $C_N(a, b)$. Pogosto se namreč zgodi, da je cena ekvivalentne arhitekture $1+d'$ glede na ceno arhitekture $a + b$ velika. Na primer, arhitektura $C_{101}(3, 5)$ je cenejša od ekvivalentne arhitekture $C_{101}(1, 32)$.

V nadaljevanju bo prikazano, kako močno vpliva povprečna razdalja med celicami na uspešnost izvajanja globalnega problema in ali je mogoče z evolucijo priti do uspešne arhitekture, ki hkrati izkazuje nizko ceno povezovanja.



Slika 2.30: Povprečna razdalja med celicami za $C_N(1, d')$ kot funkcija parametra d', prikazana le za $d' < N/2$ zaradi simetrije pri $d' = N/2$, $N = 101$. Vir: [10].

2.6.1 Fiksne arhitekture

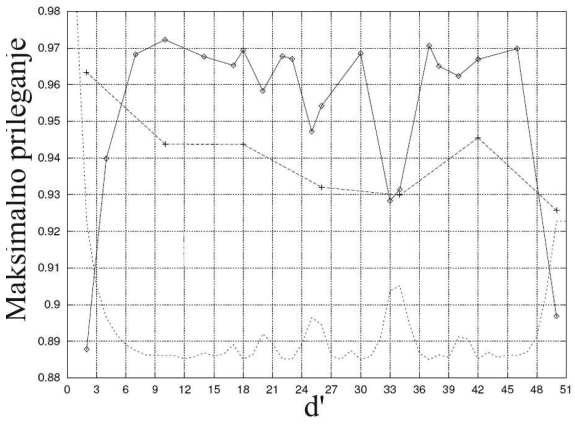
Najprej bodo podane ugotovitve, dobljene na osnovi evolucije omrežnih avtomatov s fiksnimi arhitekturami tipa $C_N(1, d')$ za vse celice, pri različnih vrednostih d', predvsem v zvezi s problemom gostote in polnjenja področij. Pri slednjem gre za spremembo stanj v mešanih področjih (celice v različnih stanjih) v stanje samih enic.

Prva ugotovitev je, da omrežni avtomat izkazuje uspešno reševanje problema gostote pri takšnih vrednostih parametra d', ki ustrezajo majhni povprečni razdalji in obratno, reševanje ni uspešno pri arhitekturah s parametri d', ki ustrezajo velikim povprečnim razdaljam med elementi omrežnega avomata (slika 2.31). Poleg tega se izkaže, da je reševanje problema polnjenja linij najboljše za minimalno vrednost d' in da se z njenim povečevanjem slabša (slika 2.32).

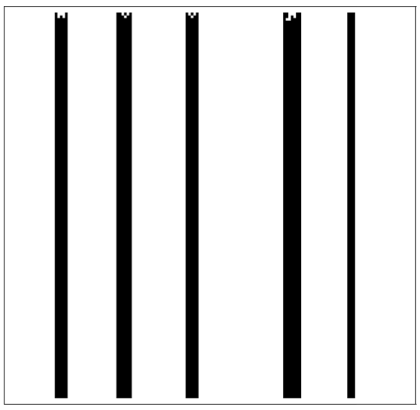
Druga ugotovitev pa pravi, da uspešnost reševanja pada linearno s povprečno razdaljo med elementi. Primer je prikazan na sliki 2.33.

Tabela 2.1: Prikaz uspešnosti arhitektur $C_N(a, b)$ na problemu gostote, 15 evolucij s 50.000 začetnimi konfiguracijami na tek. Vir: [10].

(a, b)	N	povprečna razdalja med celicami	d'	povprečno maksimalno prileganje
$(3, 5)$	101	5,98	32	0,96
$(3, 5)$	102	6,02	21	0,96
$(3, 6)$	101	13	2	0,88
$(3, 6)$	102	ni povezav	ne obstaja	0,75



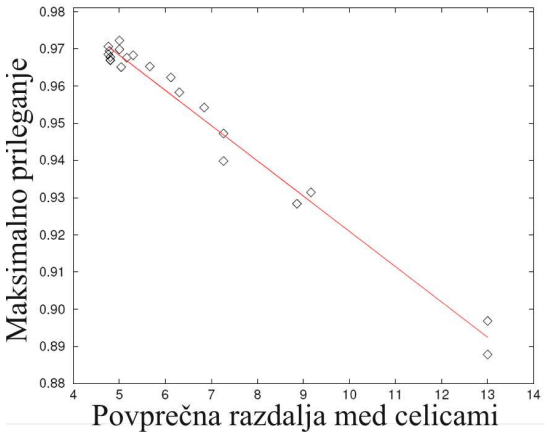
Slika 2.31: Maksimalno prileganje, dobljeno s postopkom CPA, na problemu gostote (zgoraj) in problemu polnenja linij (v sredini), povprečeno preko 420 tekov in povprečna razdalja med celicami (spodaj) v odvisnosti od d'. Za vsak d' je bilo izvedenih 15 evolucij s 50.000 konfiguracijami na tek. Vir: [10].



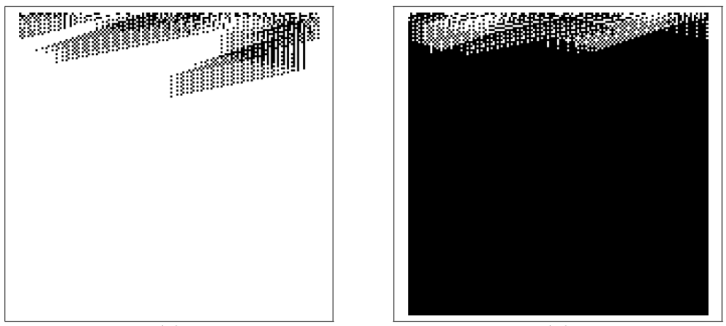
Slika 2.32: Primer polnjenja linij, prikaz nehomogenega celičnega avtomata dobljenega s postopkom CPA, $N = 149,\ r = 2,\ C_{149}(1, 2)$. Vir: [10].

Kot prikazuje tabela 2.1, je obnašanje celičnih avtomatov močno odvisno od arhitekture. Poleg tega se izkaže, da sprememba sosednosti glede na regularno celično arhitekturo izboljša rezultate procesiranja netrivialnih globalnih problemov. Lokalni in globalni problemi imajo različne arhitekture, pri katerih so rešitve uspešne.

Na sliki 2.34 je prikazano procesiranje nehomogenega celičnega avtomata po postopku CPA z arhitekturo celic $C_{149}(3, 5)$.



Slika 2.33: Maksimalna uspešnost celičnega avtomata na problemu gostote v odvisnosti od povprečne celične razdalje, $C_{101}(1, d')$, ki izkazuje očitno linearno regresijo. Vir: [10].

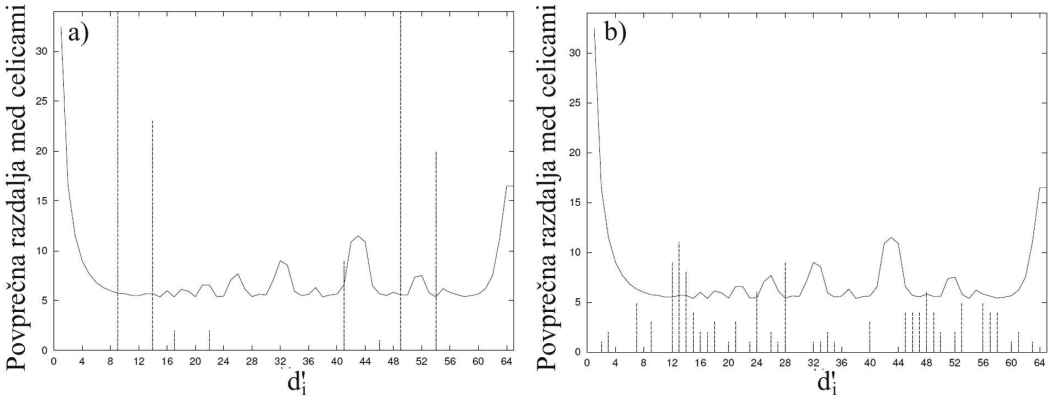


Slika 2.34: Procesiranje nehomogenega celičnega avtomata po postopku CPA z arhitekturo celic $C_{149}(3, 5)$: a) začetna gostota enic je 0,48, b) začetna gostota enic je 0,51. Vir: [10].

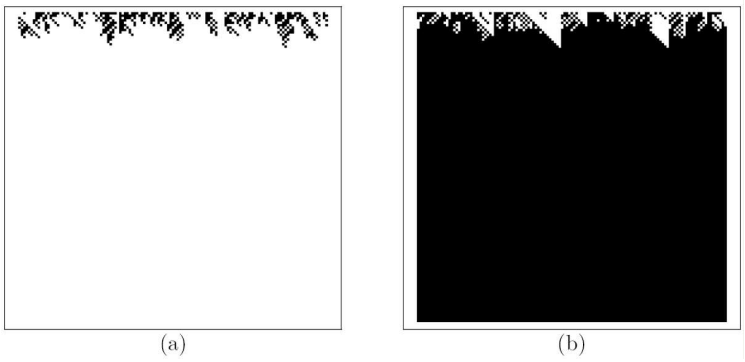
2.6.2 Razvijajoče arhitekture

S pomočjo spremenjenega postopka CPA bodo končno podani tudi rezultati evolucije, v kateri se bodo spreminjala hkrati pravila prehajanja stanj celic, kot tudi njihove povezave z ostalimi celicami. Rezultat bo zato nehomogeno omrežje avtomatov, pri katerem bo v splošnem lahko vsaka celica drugačna glede na njeno pravilo in povezave. Genotip bo zato sestavljen iz dveh delov. Prvi bo enak kot prej (tabela prehajanja stanj v odvisnosti od vplivnih celic), drugi pa bo vseboval parametra (a_i, b_i) oziroma parameter d'_i za vsako celico $i = 0, \ldots, N-1$. Ker se je izkazalo, da so rezultati boljši, če je pogostost spreminjanja pravil večja od pogostosti spreminjanja povezav, je bilo izvedeno spreminjanje pravil vsakih $C = 300$ konfiguracij in spreminjanje povezav vsakih $C' = 1.500$ začetnih naključnih konfiguracij.

Rezultati tako spremenjenega postopka CPA na problemu gostote kažejo na naslednje značilnosti: razvita arhitektura ustreza v večini primerov minimalni povprečni razdalji, rezultati procesiranja pa so boljši od rezultatov, dobljenih s fiksnimi arhitekturami (sliki 2.35 in 2.36).



Slika 2.35: Evolucija arhitektur $C_{129}(1, d_i')$ za dva teka. Prikazan je histogram (črtkano) in povprečna razdalja med celicami za problem gostote. Uspešnost v obeh primerih je enaka 0,98, povprečna razdalja pa je za primer a) 31,5 in za primer b) 30,8. Vir: [10].



Slika 2.36: Procesiranje celičnega avtomata na problemu gostote z nehomogeno arhitekturo $C_{129}(1, d_i')$: a) začetna gostota enic je 0,496, b) začetna gostota enic je 0,504. Hitrost je večja kot pri celičnem avtomatu s fiksno arhitekturo oziroma pri homogenem celičnem avtomatu s pravilom GKL. Vir: [10].

Razvijajoče nizko cenovne arhitekture

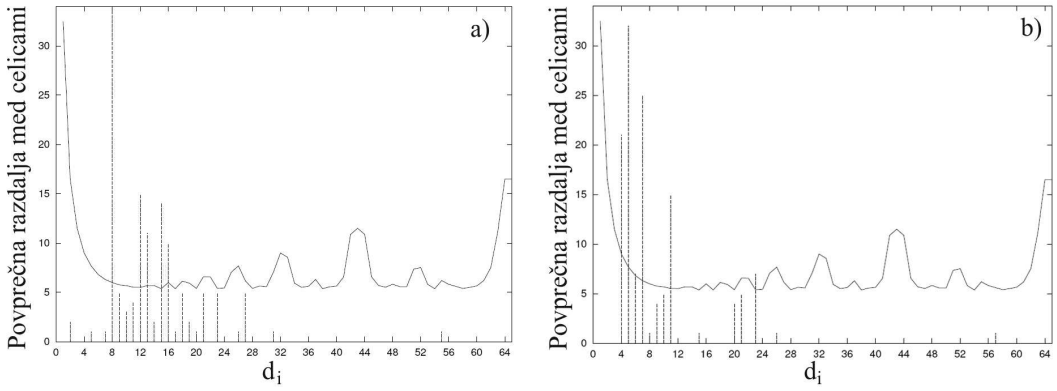
V primeru, ko je poleg uspešno razvitega omrežnega avtomata cilj dobiti tudi nizko cenovno arhitekturo, to je rešitev, ki ima nizko ceno povezovanja na celico, ki je

definirana z d'_i oziroma z $a_i + b_i$, je potrebno spremeniti kriterijsko funkcijo:

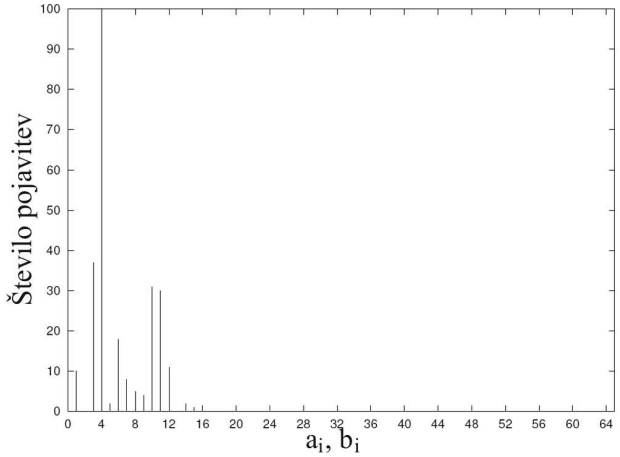
$$f'_i = f_i - \rho \cdot d'_i/N \qquad \text{oziroma} \qquad f'_i = f_i - \rho \cdot (a_i + b_i)/N \quad , \tag{2.5}$$

v kateri koeficient ρ predstavlja ceno povezovanja.

Pri reševanju problema gostote je koeficient ρ smiselno nastaviti v območju med 0,02 in 0,04. Rezultat evolucije je nova arhitektura, ki je cenejša od predhodne ob minimalni degradaciji uspešnosti (sliki 2.37 in 2.38).



Slika 2.37: Evolucija nizkocenovne arhitekture $C_{129}(1, d'_i)$. Poleg povprečne razdalje med celicami je prikazan še histogram (črtkano). a) $R = 0{,}97$, srednja vrednost $d'_i = 13{,}6$, b) $R = 0{,}96$, srednja vrednost $d'_i = 9$. Vir: [10].



Slika 2.38: Prikaz rezultata tipičnega teka evolucije z uporabo $C_{129}(a_i, b_i)$, kjer histogram podaja število pojavov a_i, b_i v omrežnem avtomatu. $R = 0{,}97$, srednja vrednost $a_i + b_i = 6{,}1$. Vir: [10].

2.7 Modeliranje s celičnimi avtomati

Celični avtomati so posebno primerni za modeliranje dinamičnih sistemov. Takšno modeliranje je pomembno, kadar ne poznamo dinamskih enačb sistema, temveč so nam na razpolago le empirični (izmerjeni) podatki o določenih spremenljivkah sistema. Vzemimo primer, da poznamo dinamiko dvo-dimenzionalnega sistema in sicer tako, da poznamo njegove izhodne spremenljivke v zaporednih diskretnih časih. Zanima pa nas, kako se sistem obnaša v prihodnosti. Če gre za stacionarne razmere, potem predpostavljamo, da lahko iz zgodovine obnašanja sistema sklepamo na njegovo prihodnost. Vendar pa potrebujemo način, da njegovo zgodovino primerno upoštevamo tako, da je na njeni osnovi mogoča napoved prihodnjega obnašanja sistema.

Za takšno nalogo so zelo primerni celični avtomati v kombinaciji z evolucijskim algoritmom, kot sta na primer postopek CPA ali genetski algoritem, ki omogočata določitev funkcij in/ali topologije celic, na osnovi poznanih empiričnih podatkov o sistemu v diskretnih zaporednih časih. Pri tem je pomembna tudi izbira kriterijske funkcije, ki spremlja postopek evolucijskega iskanja funkcijskih pravil in/ali sosednosti celic v celičnem avtomatu.

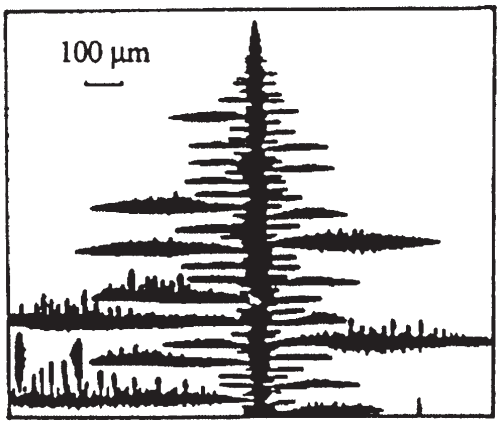
V nadaljevanju bo podan primer modeliranja dinamičnega procesa rasti kristala NH_4Br na osnovi evolucije uniformnega celičnega avtomata.

2.7.1 Modeliranje rasti kristala s celičnim avtomatom

Rast kristala NH_4Br izkazuje kompleksne oblike skozi čas. Če z dvo-dimenzionalnimi slikami podajamo stanje kristala v zaporednih diskretnih časih, potem takšno zaporedje slik poenostavljeno podaja empirične podatke o rasti omenjenega kristala. Glede na to, da seveda ni mogoče podati analitičnega opisa takšnega dinamičnega sistema, se postavlja vprašanje, ali je mogoče s sintezo celičnega avtomata na osnovi evolucijskega algoritma priti do relevantnih vhodov celičnega avtomata, ki omogočajo zapis dinamike rasti kristala.

Eksperimentalni zajem podatkov

Tipično sliko kristala NH_4Br podaja slika 2.39. Njegova značilnost so številni izrastki ali dendriti, ki tvorijo precej kompleksno strukturo. Video slike kristala v različnih časih kažejo na njegovo stanje razvoja (rast), vendar te slike ne podajajo dodatnih podatkov, kot je temperatura, ali drugih vplivnih veličin v posameznih časovnih trenutkih. Simetrija dendritov kristala omogoča prikaz dinamike kristala z zaporedjem dvo-dimenzionalnih slik. Ker je pri tem pomembno le, kateri del snovi

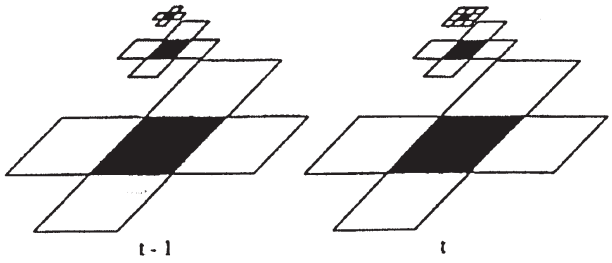


Slika 2.39: Tipično stanje v razvoju kristala NH_4Br. Rast dendritov poteka od leve proti desni. Vir: [13].

je v trdem in kateri v tekočem stanju, imamo opraviti s sliko binarnih točkovnih elementov, kjer 0 (bela točka) pomeni tekoči del, 1 (črna točka) pa trdi del.

Opis meritev je podrobneje podan v [13]. Podatke o posameznih slikah je posnela kamera skozi mikroskop. Po digitalni obdelavi signalov je bila vsaka slika kristala predstavljena s 120 x 120 točkami. Pogostost vzorčenja 2 sliki/sekundo je bila prilagojena rasti kristala.

Da se pri določanju časovne dinamike celic celičnega avtomata upoštevajo različne časovne skale in različne krajevne skale, so za relevantne sosednostne celice izbrane celice iz časovnega okvirja $t-1$ in celice iz časovnega okvirja t. Ker je realno pričakovati, da se vpliv celic z oddaljenostjo manjša, je za poln nabor potencialno vplivnih spremenljivk ali glavni nabor izbranih 28 binarnih spremenljivk, ki so prikazane na sliki 2.40. Najmanjši kvadratki so direktni binarni predstavniki slikovnih



Slika 2.40: Glavni nabor potencialnih vplivnih spremenljivk. Vir: [13].

točk, večje pa je potrebno v binarne spremenljivke šele transformirati na osnovi večinskega principa. Na primer, če je več točk v nekem področju belih, je tudi binarna spremenljivka, ki v glavnem naboru predstavlja to področje, bela. Črni kvadratki na sliki so celice, ki jih pri tem ne upoštevamo. Vseh potencialnih vpliv-

nih binarnih spremenljivk je torej 28, 12 v časovnem okviru $t-1$ in 16 v okviru t. Iz tega nabora je potrebno določiti najbolj informativne spremenljivke, ki izkazujejo največjo korelacijo med vhodnimi (vplivnimi) spremenljivkami in izhodnimi (odvisnimi) stanji celic v celičnem avtomatu.

Genetski algoritem za sintezo celičnih avtomatov

Iščemo homogeno pravilo za celični avtomat, ki opisuje dinamiko rasti kristala, kot to izhaja iz eksperimentalnih podatkov. V ta namen je potrebno najprej določiti, katere spremenljivke iz sosednosti najbolje določajo prihodnja stanja. Potem, ko so relevantne spremenljivke določene, je mogoče zgraditi homogeno stohastično pravilo na osnovi histogramov vhodno izhodnih kombinacij celic celičnega avtomata. S tem pravilom lahko celični avtomat stohastično napoveduje naslednja stanja rasti kristala.

Najbolj informativne celice v sosednostnem področju je mogoče določiti s pomočjo genetskega algoritma. Pri tem tvorijo nabori, ki jih dobimo iz glavnega nabora z izpuščanjem posameznih spremenljivk, populacijo, nad katero izvajamo operacije evolucije (selekcijo, križanje in mutacijo) toliko časa, dokler ustrezna kriterijska funkcija ne doseže zadovoljive vrednosti. Za vsak nabor se na podlagi eksperimentalnih podatkov določi verjetnosti vhodnih in izhodnih kombinacij, na podlagi katerih je možna ocena kvalitete nabora. Ker ima glavni nabor 28 bitov, je število (pod)naborov enako 2^{28}, kar je tako velika številka, da je uporaba genetskega algoritma praktično edina možnost, da pridemo do rezultata.

Kriterijska funkcija

Kot smo videli, je mogoče za vsak nabor določiti iz eksperimentalnih podatkov verjetnostno pravilo uniformnega celičnega avtomata. Potreben je še kriterij, po katerem posamezne nabore ocenjujemo, da bi dobili najboljši nabor. V ta namen je mogoče uporabiti povprečno medsebojno informacijo, ki je definirana na osnovi ustreznih verjetnosti:

$$I(X,Y) = \sum_i \sum_j P(x_i, y_j) \cdot \log_2 \frac{P(x_i, y_j)}{P(x_i) \cdot P(y_j)} \quad , \tag{2.6}$$

kjer je x_i i-ta kombinacija vhodnega (binarnega) vektorja X, torej nabora in y_j j-ta vrednost celice v naslednjem časovnem okviru Y, $P(X,Y)$ je vezana verjetnost dogodkov X in Y, $P(X)$ in $P(Y)$ pa sta marginalni verjetnosti spremenljivk X in Y. Povprečno medsebojno informacijo lahko ekvivalentno definiramo tudi na osnovi

entropije:

$$\begin{aligned} I(X,Y) &= H(X) + H(Y) - H(X,Y) \\ H(X) &= -\sum_i P(x_i) \cdot \log_2 P(x_i) \end{aligned} \tag{2.7}$$

Če je medsebojna informacija med naborom in stanjem v naslednjem časovnem trenutku splošne celice celičnega avtomata velika, je nabor informacijsko pomemben, sicer pa ni. Torej mora biti kriterijska funkcija direktno proporcionalna povprečni medsebojni informaciji. Zaradi nevarnosti, da pride do prevelikega upoštevanja eksperimentalnih podatkov (*ang.* overfitting), je potrebno vzeti za kriterijsko funkcijo izraz:

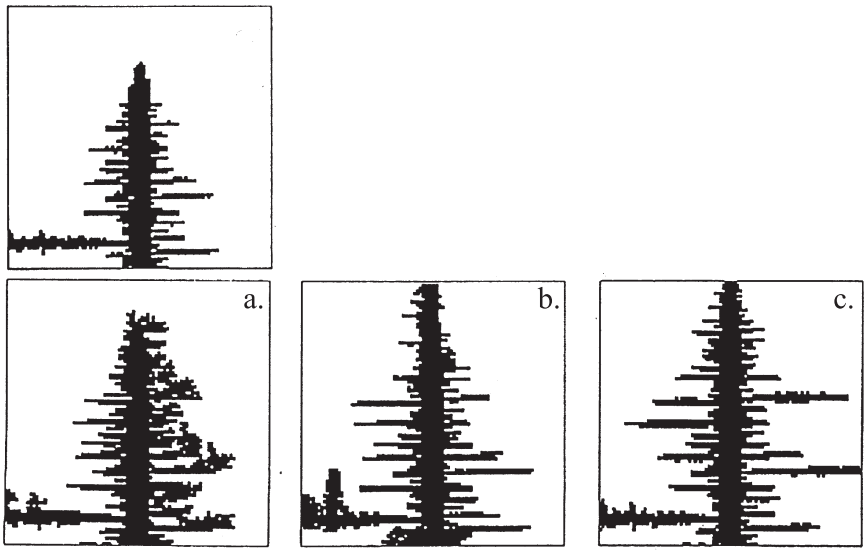
$$F = I(x,y) - \frac{2^m}{N} \qquad , \tag{2.8}$$

kjer je m število bitov v naboru, N število eksperimentalnih podatkov, x je stanje nabora, y pa vrednost točke v naslednjem trenutku.

Rezultati

Rezultati, ki so podani v nadaljevanju, izhajajo iz 10^3 podatkovnih posnetkov kristala NH_4Br v različnih, a zaporednih časih njegove rasti, vzorčenih s hitrostjo 2 slik na sekundo. Na vsaki sliki se upošteva samo tiste točke, ki imajo stanje 0 (bela, tekoče stanje), vsaj ena od njenih štirih najbližjih sosedov pa je 1 (črna, trdno stanje). Takih točk je približno 800 na vsaki sliki. Pri računanju verjetnosti se tako upošteva približno 8×10^5 točk. Ob koncu delovanja genetskega algoritma, ko z naraščanjem spremenljivk v naboru vrednost F začne padati, ima 10 končnih naborov približno isto vrednost kriterijske funkcije F. Vsak od njih vsebuje 13 od 28 spremenljivk in sicer tako, da skupaj pokrivajo skoraj vse spremenljivke glavnega nabora. Zanimivo je tudi, da nobena spremenljivka ni bila hkrati izbrana v obeh časovnih slikah (t in $t-1$).

Slika 2.41 zgoraj prikazuje začetno stanje slike kristala, slika 2.41 spodaj pa prikazuje tri različne vzorce v istem časovnem oknu ($t = 72$), ki izhajajo iz začetnega stanja. Prvi a) izhaja iz začetnih generacij algoritma, kjer se pravila celičnega avtomata še niso dobro prilegala eksperimentalnim podatkom (nizka vrednost F), vzorec b) podaja najboljši nabor glede kriterijske funkcije, vzorec c) pa prikazuje dejansko stanje kristala po $t = 72$ časovnih korakih. Velja splošna ocena, da je glede na specifično časovno in krajevno odvisnost stanj celic reprezentativnega celičnega avtomata, rezultat zelo dober v kvalitativnem smislu, manj pa v smislu natančnosti oziroma glede srednjega kvadrata napake med dejansko in modelirano vrednostjo.



Slika 2.41: Začetno stanje kristala (zgoraj) in stanje kristala po $t = 72$ časovnih korakih (spodaj): a) neprilagojen celični avtomat, b) najbolj prilagojen celični avtomat in c) dejansko stanje kristala. Vir: [13].

Opisani postopek je splošen in je osnovan na iskanju verjetnostnega celičnega avtomata z evolucijskim postopkom na osnovi eksperimentalnih podatkov (histogramov oziroma verjetnosti). Kriterijska funkcija je pri tem premo sorazmerna s povprečno medsebojno informacijo med vhodnimi in izhodnimi spremenljivkami oziroma njihovimi porazdelitvami.

Poglavje 3

Paralelni računalniki

3.1 Uvod

Paralelni računalnik je računalniški sistem z več procesorji, ki podpira paralelno programiranje [16, 17]. Paralelne računalnike delimo v dve veliki skupini, v večračunalniške sisteme in v centralizirane večprocesorske sisteme.

Večračunalniški sistem je paralelni računalnik, sestavljen iz množice računalnikov in povezovalnega omrežja. Računalniki komunicirajo med seboj s pošiljanjem in sprejemanjem sporočil.

Centralizirani večprocesorski sistem (tudi simetrični multiprocesor) je višje integriran sistem v katerem si vse centralne procesne enote delijo skupni globalni pomnilnik. Ta deljeni pomnilnik podpira komunikacijo in sinhronizacijo med procesorji.

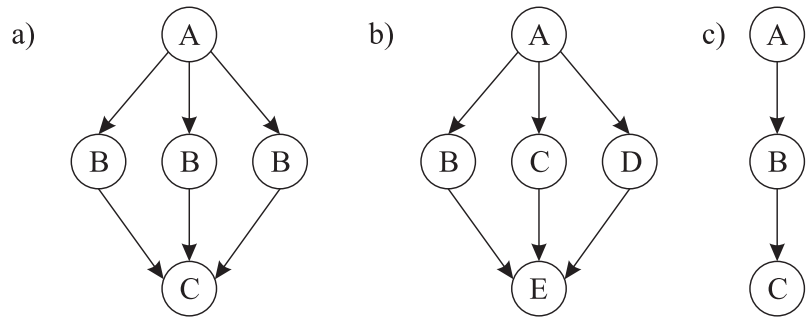
Paralelno programiranje je programiranje v jeziku, ki omogoča eksplicitno označevanje deležev računanja, ki se lahko izvajajo istočasno na različnih procesorjih. Zaenkrat še ne obstajajo komercialni in uspešni paralelni prevajalniki, ki bi prevedli standardne sekvenčne programe v kode, ki bi se izvajale na paralelnih računalnikih s številnimi procesorji. Ena od alternativ je, da napišemo svoj paralelni program. Druga možnost pa je uporaba tehnologije GRID z ustreznimi programskimi okolji, ki poskrbijo za distribucijo obstoječe sekvenčne aplikacije (podatkov in funkcijskih modulov) v paralelni računalnik. V nadaljevanju si bomo ogledali obe možnosti.

MPI (*ang.* Message Passing Interface) je standardna specifikacija za knjižnico funkcij, pri kateri komunikacija med procesorji sloni na pošiljanju sporočil, in je brezplačno dosegljiva na Internetu. V primeru centraliziranih večprocesorskih sistemov, ko si procesorji delijo glavni pomnilnik, je knjižnica OpenMP boljši način za povezovanje procesorjev med seboj. Če imamo več centraliziranih večprocesorskih sistemov

povezanih v omrežje, potem je pri pisanju paralelnih programov primerna uporaba hibridnih programov, ki uporabljajo tako knjižnico MPI kot knjižnico OpenMP.

3.2 Odkrivanje sočasnosti

Formalni način za prikaz potencialnih sočasnosti procesiranja je v podatkovno odvisnem grafu (*ang.* Data Dependence Graph ali kratko DDG), ki je usmerjen graf, v katerem vozlišča predstavljajo naloge, povezave med njimi pa določajo odvisnost. Na primer, povezava, usmerjena od vozlišča u do vozlišča v, pomeni, da se mora naloga u izvesti pred nalogo v. Če med dvema nalogama ni povezave, sta neodvisni in se lahko izvedeta sočasno. Slika 3.1 prikazuje tri različne vzorce nalog in ustrezne DDG.



Slika 3.1: Primeri paralelizmov z DDG: a) podatkovni paralelizem, b) funkcijski paralelizem in c) sekvenčni graf.

Na sliki 3.1a je prikazan DDG, ki ustreza podatkovnemu paralelizmu. Naloga B se izvaja paralelno na treh podatkovnih segmentih. Primer fino-zrnatega (*ang.* fine-grained) podatkovnega paralelizma v sekvenčnem algoritmu podaja koda na sliki 3.2. Gre za operacijo seštevanja, ki se lahko izvede sočasno za vseh 100 elementov polj b in c in se shrani v polje a.

```
FOR i := 0 to 99 DO
    a[i] := b[i] + c[i]
END FOR
```

Slika 3.2: Sekvenčni program, v katerem je očiten podatkovni paralelizem.

Slika 3.1b podaja DDG, ki izkazuje funkcijski paralelizem, saj se neodvisne naloge (B, C, D) izvajajo sočasno nad različnimi podatkovnimi segmenti. Primer fino-zrnatega funkcijskega paralelizma v sekvenčnem algoritmu podaja koda na sliki 3.3.

V tem primeru sta tretji in četrti stavek odvisna izključno od spremenljivk a in b in se zato lahko izvedeta hkrati.

```
a := 2
b := 3
m := (a + b) / 2
s := (a * a + b * b) / 2
v := s - m * m
```

Slika 3.3: Sekvenčni program v katerem je očiten funkcijski paralelizem.

Kot je razvidno iz slike 3.1c, sekvenčni DDG ne dovoljuje paralelizma, če gre za enkratno izvajanje programa. V primeru večkratnega izvajanja istega programa pa je mogoče govoriti o cevovodnem procesiranju, pri katerem se sekvenčne naloge izvajajo hkrati, vendar nad različnimi podatki. Cevovodna shema je analogna proizvodni liniji, kjer se neprestano ponavljajo iste naloge nad različnimi objekti.

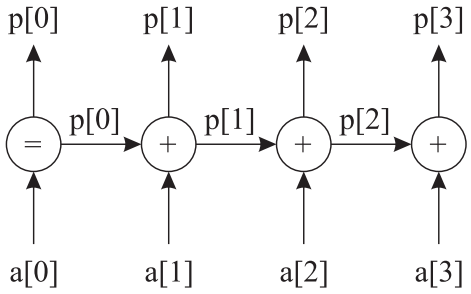
Na sliki 3.4 je prikazano, kako je mogoče programsko zanko za računanje delnih vsot razviti v enakovredno cevovodno shemo. Ustrezno cevovodno strukturo prikazuje slika 3.5.

```
p[0] := a[0]
FOR i := 1 TO 3 DO
    p[i] := p[i-1] + a[i]
END FOR

p[0] := a[0]
p[1] := p[0] + a[1]
p[2] := p[1] + a[2]
p[3] := p[2] + a[3]
```

Slika 3.4: Sekvenčni program primeren za cevovodni paralelizem zgoraj in enakovredna cevovodna shema spodaj.

Primer: Tvorjenje gruč podatkov Tvorjenje gruč podatkov (*ang.* data clustering) je tipičen računsko zahteven problem, ki se pogosto uporablja v številnih aplikacijah. Zanima nas, ali obstajajo možnosti za njegovo paralelno realizacijo.

Vzemimo, da moramo zbirko N dokumentov razvrstiti v D različnih kategorij glede na njihovo vsebino. Dokumente želimo razvrstiti v K gruč glede na sorodnost vsebin tako, da bo izbrana kriterijska funkcija, ki določa, kako razvrščati dokumente, čim



Slika 3.5: Cevovodna shema za računanje delnih vsot.

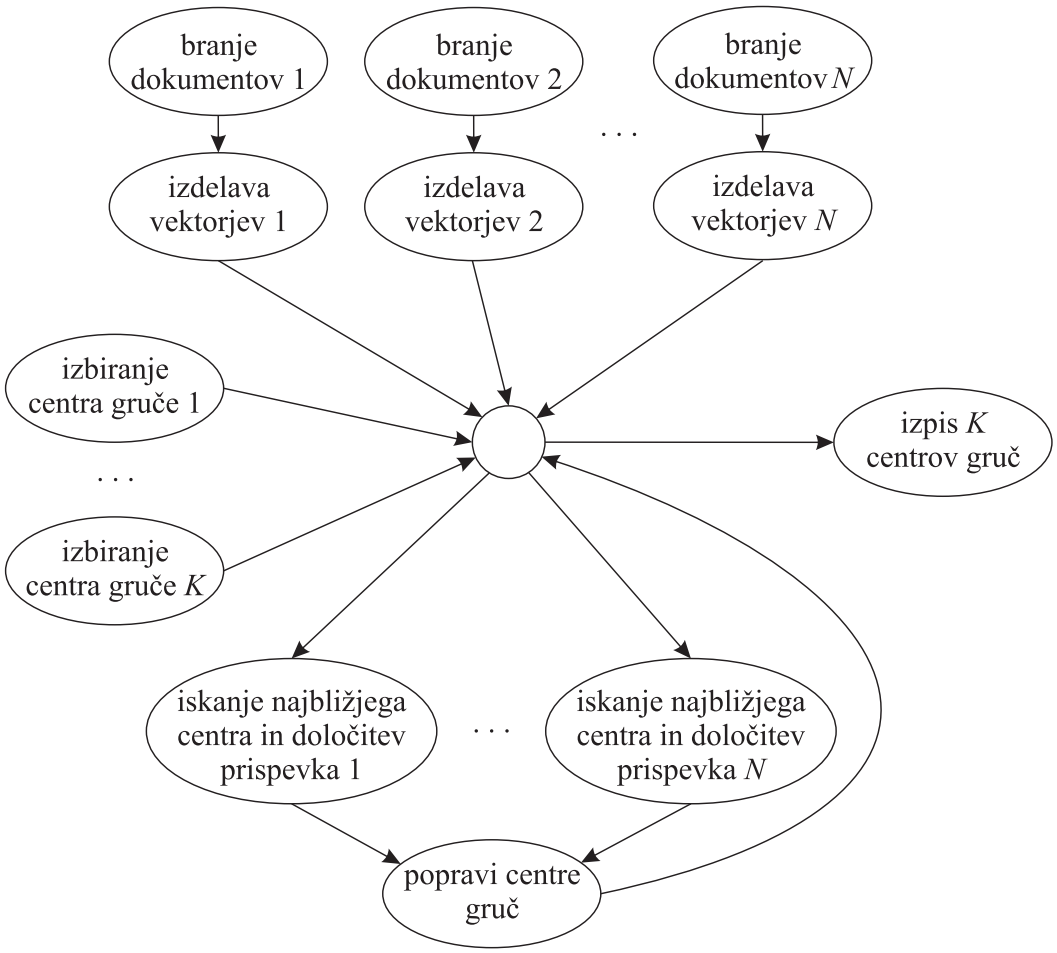
bolj optimalna. Sekvenčni algoritem za razvrščanje dokumentov sestavljajo naslednji koraki:

- Preberi N dokumentov
- Za vsak dokument izdelaj D dimenzionalni vektor, ki opisuje zastopanost vsebin
- Izberi K začetnih centrov gruč, v katere so dokumenti razvrščeni naključno
- Ponavljaj naslednje korake do izpolnitve pogoja za zaustavitev:
  - Za vsak dokument poišči najbližji center gruče in določi prispevek k kriterijski funkciji
  - Popravi položaje K centrov tako, da izboljšaš kriterijsko funkcijo
- Izpiši K centrov gruč.

Najprej nas zanima DDG. Čeprav bi lahko za vsak korak algoritma uporabili eno vozlišče, je bolje vozlišča grafa vezati na vsak korak algoritma za vsak dokument ali vsak center gruče, saj dobimo tako več možnosti za snovanje paralelnih algoritmov. Ustrezni DDG je prikazan na sliki 3.6. Dober DDG omogoča relativno enostavno določitev podatkovnega in funkcijskega paralelizma.

V DDG za problem razvrščanje dokumentov v gruče lahko opazimo naslednje možnosti za uporabo podatkovnega paralelizma:

- vse dokumente lahko preberemo naenkrat,
- hkrati se lahko generirajo vsi dokumentni vektorji,
- začetne centre gruč lahko generiramo hkrati in
- hkrati se lahko za vsak dokumentni vektor računajo najbližji center gruče in doprinos vektorja k kriterijski funkciji.



Slika 3.6: DDG za razvrščanje dokumentov v gruče. Osrednje prazno vozlišče je dodano za večjo preglednost.

Glede funkcijskega paralelizma v DDG na sliki 3.6 pa velja, da sta edina neodvisna niza vozlišč tista, ki predstavljata branje dokumentov skupaj z izdelavo dokumentnih vektorjev in opravila, ki generirajo začetne centre gruč.

3.3 Programiranje paralelnih računalnikov

Obstajajo različni pristopi, na osnovi katerih je mogoče razvijati aplikativno programsko opremo za paralelne računalnike. Najbolj poznani med njimi so opisani v nadaljevanju.

Razširitev prevajalnika z novimi ukazi Ena možnost je razvoj prevajalnika, ki bi zaznal in nakazal možnost za paralelno izvedbo programa v sekvenčnih programskih jezikih. Čeprav je bilo opravljenega veliko dela v tej smeri, predvsem na jeziku Fortran, pa komercialnih in uspešnih rezultatov še ni.

Razširitev sekvenčnih programskih jezikov To je mnogo bolj konzervativna pot, ki klasičnim programskim jezikom dodaja funkcije, s katerimi je mogoče kreirati in zaključevati procese, jih sinhronizirati in omogočati komunikacijo med njimi. V tem primeru je potrebno razločevanje med globalnimi spremenljivkami, ki si jih procesi delijo, in privatnimi spremenljivkami, ki so last vsakega procesa posebej.

Takšna razširitev je najlažja, najhitrejša, najcenejša in ravno zato najbolj popularna. Zahteva le razvoj knjižnice funkcij oziroma podprogramov, s katerimi je mogoča uporaba istega sekvenčnega programskega jezika in njegovega prevajalnika. Na primer, knjižnice, ki podpirajo standard MPI, obstajajo za vse pomembnejše procesorje. Paralelni programi so zato na nivoju programske kode prenosljivi.

Pri uporabi funkcij in podprogramov, ki omogočajo pralelno programiranje, ima programer visoko stopnjo fleksibilnosti, saj je mogoče paralelni program napisati na več različnih načinov. Ker pri tem pristopu prevajalnik ni vključen v generiranje paralelne kode, tudi ne more detektirati napak. Vendar pa to v praksi ne predstavlja problema, saj je tovrstno programiranje relativno enostavno.

Dodajanje paralelnega programskega nivoja Predstavljamo si, da ima paralelni program dva nivoja. Spodnji nivo vsebuje računsko jedro, ki obdeluje podatke in generira rezultate. Ta nivo ustreza zaporednemu programskemu jeziku. Zgornji nivo pa nadzira ustvarjanje in sinhronizacijo procesov ter delitev podatkov med procese. Te aktivnosti so lahko opisane v paralelnem programskem jeziku. Prevajalnik mora biti sposoben prevesti takšen dvonivojski program v kodo, ki se izvaja na paralelnem računalniku. Znana primera tega pristopa sta sistema CODE (Computationaly Oriented Display Environment) in HENCE (Heterogeneous Network Computing Environment). Oba omogočata uporabniku, da prikaže paralelni program kot usmerjeni graf, kjer vozlišča predstavljajo sekvenčne procedure, usmerjene povezave pa podatkovno odvisnost med procedurami. Žal zaradi specifičnih znanj v tej smeri še ni komercialnih produktov.

Paralelni jeziki Ta možnost omogoča programerju, da paralelne operacije izraža eksplicitno. Slednje je možno z razvojem novega paralelnega jezika (na primer OCCAM), ali pa z dodajanjem paralelnih konstruktov k obstoječim sekvenčnim jezikom (na primer Fortran 90, High Performance Fortran, C^*).

Čeprav poteka razvoj vseh omenjenih različic nezadržno naprej, pa je paralelno

programiranje v jeziku C z uporabo knjižnic MPI in/ali OpenMP najpopularnejši pristop k paralelnemu programiranju.

3.4 Arhitekture paralelnih sistemov

Zadnjih trideset let prejšnjega stoletja je zaznamovalo intenzivno raziskovanje arhitektur paralelnih računalniških sistemov. Nekatere raziskave so šle v smeri izdelovanja namenskih procesorjev za paralelne računalnike v tehnologiji VLSI (*ang.* Very Large Scale Integration). V drugih so uporabljali univerzalne procesorje, ki so bili uporabljani v delovnih postajah in osebnih računalnikih. Pomembno vprašanje je bilo, ali naj vsebuje paralelni računalnik nekaj deset visoko zmogljivih namenskih procesorjev ali tisoče manj zmogljivih univerzalnih procesorjev.

Danes so sistemi z nekaj tisoč univerzalnimi procesorji realnost. Sodobni paralelni računalniki imajo vgrajene univerzalne procesorje, ker razvoj namenskih procesorjev ni mogel slediti napredku na področju univerzalnih komercialnih procesnih modulov.

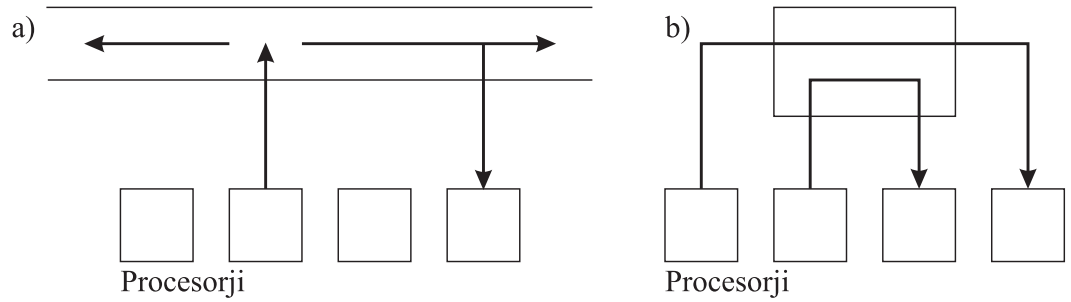
V nadaljevanju bo podan pregled arhitektur paralelnih sistemov. Začeli bomo z opisom različnih povezovalnih omrežij, ki služijo za povezovanje procesorjev znotraj paralelnega sistema. Sledila bo predstavitev polj procesorjev, večprocesorskih sistemov in večračunalniških sistemov, ki predstavljajo tri najbolj popularne arhitekture paralelnih računalnikov zadnjih dveh desetletij. Ob koncu bo podana razlika med gručo procesorjev v paralelnem računalniškem sistemu in omrežjem delovnih postaj oziroma osebnih računalnikov.

3.4.1 Povezovalni mediji

Procesorji lahko v paralelnem računalniku komunicirajo preko deljenega (*ang.* shared) ali preklopnega (*ang.* switched) povezovalnega medija.

Deljeni medij omogoča prenos enega sporočila naenkrat. Samo en procesor lahko v določenem času pošlje sporočilo, vsi ostali ga lahko le poslušajo. Sporočilo lahko berejo samo tisti procesorji, ki jim je namenjeno. Tipični predstavnik deljenega medija je Ethernet (slika 3.7a). Pred pošiljanjem sporočila po deljenem mediju mora procesor zaznati, da je medij prost. Če želi več procesorjev poslati sporočilo istočasno, pride do trka. Trkom se običajno izognemo tako, da procesorji prenos ponovijo po nekem naključno izbranem času. Trki precej degradirajo zmožnosti deljenega medija.

Preklopni mediji podpirajo prenos sporočil od točke do točke (*ang.* Point-to-point) med pari procesorjev (slika 3.7b). Vsak procesor ima svojo komunikacijsko pot do



Slika 3.7: Povezovalni mediji: a) deljeni in b) preklopni.

stikalnega medija. Stikala imajo dve pomembni prednosti pred deljenim medijem. Prvič, podpirajo sočasen prenos več sporočil med pari procesorjev in drugič, podpirajo skaliranje povezovalnega omrežja v primeru povečanega števila procesorjev. Zaradi bistvenih prednosti bo v nadaljevanju opisanih nekaj najbolj znanih preklopnih omrežij.

Preklopna omrežja

Preklopno omrežje lahko predstavimo z grafom, v katerem kvadratna vozlišča določajo procesorje, okrogla pa vozlišča stikala. Vsak procesor je povezan na eno stikalo. Stikala povezujejo procesorje in/ali ostala stikala.

Ločimo neposredne in posredne topologije. Pri neposredni topologiji je razmerje med stikalnimi in procesorskimi vozlišči enako 1:1. Vsako stikalo je povezano z enim procesorjem in z enim ali več drugimi stikali. Pri posredni topologiji je razmerje večje kot 1:1 v korist stikal, saj nekatera stikala povezujejo samo druga stikala.

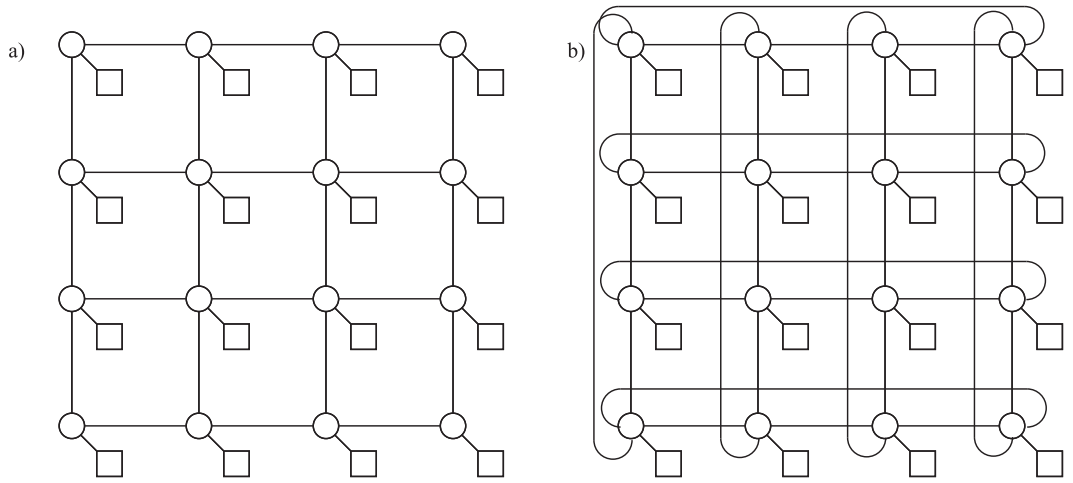
Različna preklopna omrežja je mogoče ocenjevati glede na njihove zmožnosti za implementacijo paralelnih algoritmov. Kriteriji za ocenjevanje so naslednji:

- Premer omrežja je najdaljša razdalja med dvema stikalnima vozliščema. Ker določa spodnjo mejo kompleksnosti paralelnih algoritmov za vzpostavitev komunikacije med poljubnimi pari vozlišč, je majhen premer bolj zaželen.
- Razpolovna širina (*ang.* bisection width) določa minimalno število povezav med stikalnimi vozlišči, ki jih moramo odstraniti, da razdelimo omrežje na dve polovici. Visoka vrednost je boljša, ker delitev podatkov kot posledica razpolovitve določa spodnjo mejo kompleksnosti paralelnega algoritma.
- Število povezav na stikalo - zaželeno je, da je to število konstantno in neodvisno od velikosti omrežja, ker je tedaj bolj enostavno skalirati organizacijo procesorjev z večanjem števila vozlišč.

- Najdaljša povezava - zaradi skaliranja je najbolje, če je mogoče vsa vozlišča in povezave podati v tri-dimenzionalnem sistemu tako, da je najdaljša povezava konstantna in neodvisna od velikosti omrežja.

Sledi kratek opis najbolj znanih preklopnih omrežnih topologij v luči zgornjih kriterijev. Prve štiri so uporabljene v paralelnih računalnikih, ki so na voljo na trgu, zadnji dve pa sta zanimivi za razvojne naloge.

Omrežje v obliki dvo-dimenzionalne mreže Dvo-dimenzionalna mreža (*ang.* 2D Mesh Network) je predstavnik neposredne topologije, v kateri so stikala urejena v mrežo. Komunikacija je dovoljena le med sosednimi stikali, tako kot prikazuje slika 3.8a, v primeru cikličnih povezav pa tudi s stikali na drugem robu (slika 3.8b).

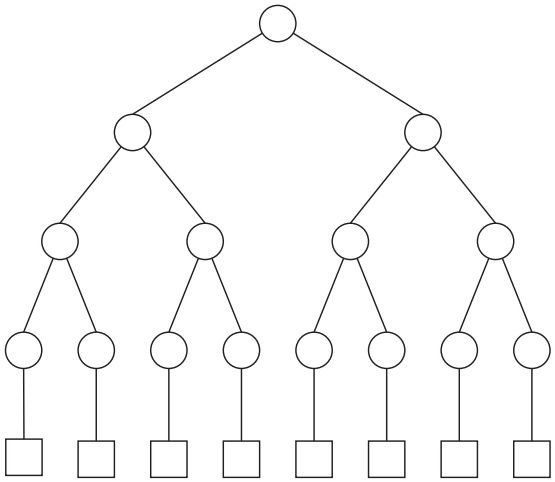


Slika 3.8: Omrežje v obliki dvo-dimenzionalne mreže a) brez cikličnih povezav in b) s cikličnimi povezavami.

V primeru n stikalnih vozlišč brez cikličnih povezav je premer minimalen, razpolovna širina pa maksimalna, če je mreža kvadratna. Tedaj sta tako premer kot razpolovna širina podana z $O(n^{1/2})$. Število povezav na stikalo je konstantno, da pa se zgraditi tudi poljubno veliko mrežo s konstantno dolžino povezav.

Omrežje v obliki binarnega drevesa Število procesorskih vozlišč v binarnem drevesu (*ang.* Binary Tree) je določeno kot $n = 2^d$, kjer je d število nivojev v drevesu. Komunikacijo med njimi omogoča $2n - 1$ stikal. Binarno drevo je prikazano na sliki 3.9. Vsak procesor je povezan na drevo kot list. Binarno drevo je predstavnik posredne topologije. Notranja stikalna vozlišča imajo največ tri povezave, dve

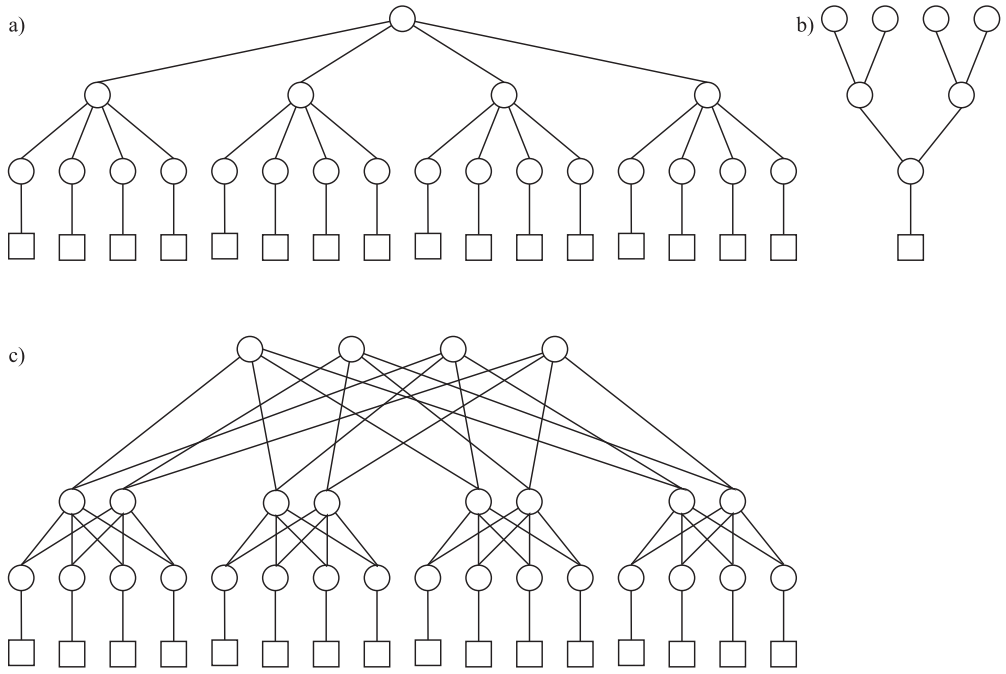
navzdol do vozlišč potomcev in eno navzgor do vozlišča staršev. Preklopno omrežje binarno drevo ima nizek premer, ki je enak $2\log_2 n$, razpolovna širina pa je najmanjša možna, torej 1. Binarnega drevesa ni mogoče urediti v tri-dimenzionalnem prostoru tako, da bi bila z naraščanjem števila vozlišč dolžina najdaljše povezave vedno manjša od izbrane konstante.



Slika 3.9: Omrežje v obliki binarnega drevesa.

Omrežje v obliki hiper-drevesa Hiper-drevo (*ang.* Hypertree Network) je predstavnik posredne topologije. Ima majhen premer in izboljšano razpolovno širino. Hiper-drevo stopnje k in globine d si najlažje predstavljamo, če ga pogledamo iz dveh različnih zornih kotov - od spredaj izgleda kot popolno drevo k-tega reda in višine d (slika 3.10a), od strani pa kot binarno drevo višine d, obrnjeno s korenom navzdol (slika 3.10b). Z združitvijo obeh pogledov dobimo kompletno omrežje. Slika 3.10c prikazuje hiper-drevo stopnje 4 in višine 2. Hiper-drevo reda 4 z globino d vsebuje 4^d procesorjev in $2^d(2^{d+1} - 1)$ preklopnih vozlišč. Njegov premer je enak $2d$, razpolovna širina pa 2^{d+1}. Število povezav na stikalno vozlišče ni nikoli večje od 6, maksimalna dolžina povezave pa je naraščajoča funkcija velikosti omrežja.

Omrežje s topologijo metulja Omrežje s topologijo metulja (*ang.* Butterfly Network) je predstavnik posredne topologija, v kateri je $n = 2^d$ procesorskih vozlišč povezanih z $n(\log_2 n + 1)$ stikalnimi vozlišči. Primer omrežja s topologijo metulja prikazuje slika 3.11. Stikalna vozlišča so razdeljena na $\log_2 n + 1$ vrstic, od katerih vsaka vsebuje n vozlišč. Vrstice so označene od 0 do $d = \log_2 n$. Vzemimo, da je s parom (i, j) označeno j-to vozlišče v i-ti vrstici, kjer je $0 \leq i \leq d$ in $0 \leq j < n$. Tedaj je vozlišče (i, j) v vrstici $i > 0$ povezano z dvema vozliščema $(i - 1, j)$ in $(i - 1, m)$, kjer je m celo število dobljeno z invertiranjem i-tega bita z leve strani v binarni predstavitvi števila j. Na primer, vozlišče $(2, 3)$ je povezano z vozliščema $(1, 3)$ in

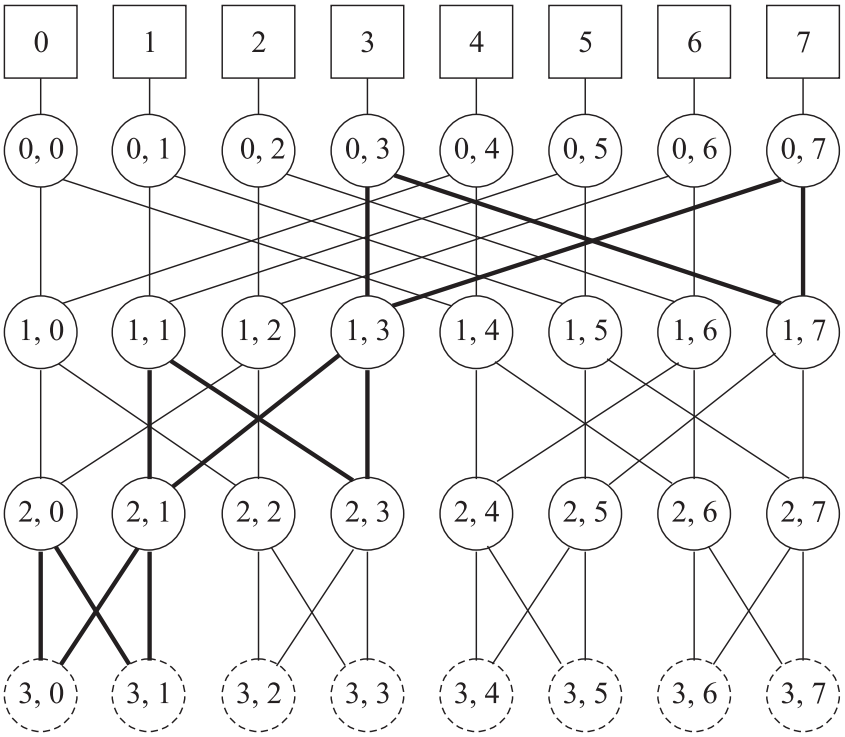


Slika 3.10: Omrežje v obliki hiper drevesa.

$(1,1)$, saj je 1 rezultat invertiranja drugega bita z leve strani v binarni predstavitvi števila $3 : 011_2 \rightarrow 001_2$). Če je vozlišče (i,j) povezano z vozliščem $(i-1,m)$, potem je vozlišče (i,m) povezano z vozliščem $(i-1,j)$. Običajno so vozlišča v vrsticah 0 in d združena tako, da oznaki $(0,j)$ in (d,j) označujeta isto vozlišče.

Z malo domišljije je mogoče opaziti strukturo metulja v cikličnem nizu povezav med vozlišči (i,j), $(i-1,j)$, (i,m), $(i-1,m)$ in (i,j). Celotno omrežje je zgrajeno iz takšnih povezav, od tod tudi ime. Z zmanjševanjem oznake vrstic se eksponentno večajo širine kril mutuljev. Zato se najdaljša povezava v omrežju povečuje s povečevanjem števila vozlišč v omrežju. V primeru, da so vozlišča v vrsticah 0 in d združena, je premer omrežja s topologijo metulja z n procesorji enak $\log_2 n$, razpolovna širina pa $n/2$.

Preprost algoritem omogoča preklapljanje stikal pri prenosu sporočil med procesorji. Pred pošiljanjem se vsako sporočilo dopolni z glavo, v kateri je binarno zakodirana številka ciljnega procesorja. Vsako stikalno vozlišče vzame vodilni (levi) bit iz glave sporočila in preklopi glede na njegovo vrednost. Če je vrednost vodilnega bita 0, potem spusti preostale bite v sporočilu po levi poti, sicer pa po desni. Vzemimo, da želi v omrežju z 8 procesorji procesor 2 poslati sporočilo procesorju 5. Procesor 2 zato pred pošiljanjem sporočila le-to dopolni z naslovom procesorja 5 (101_2). Prvo stikalno vozlišče $(0,2)$ sprejme 101+*sporočilo* in pošlje 01+*sporočilo* na desno proti stikalnemu vozlišču $(1,6)$. To pošlje 1+sporočilo naprej po levi poti proti vozlišču $(2,4)$, ki končno usmeri *sporočilo* po desni poti do vozlišča $(3,5)$, ki je sinonim za



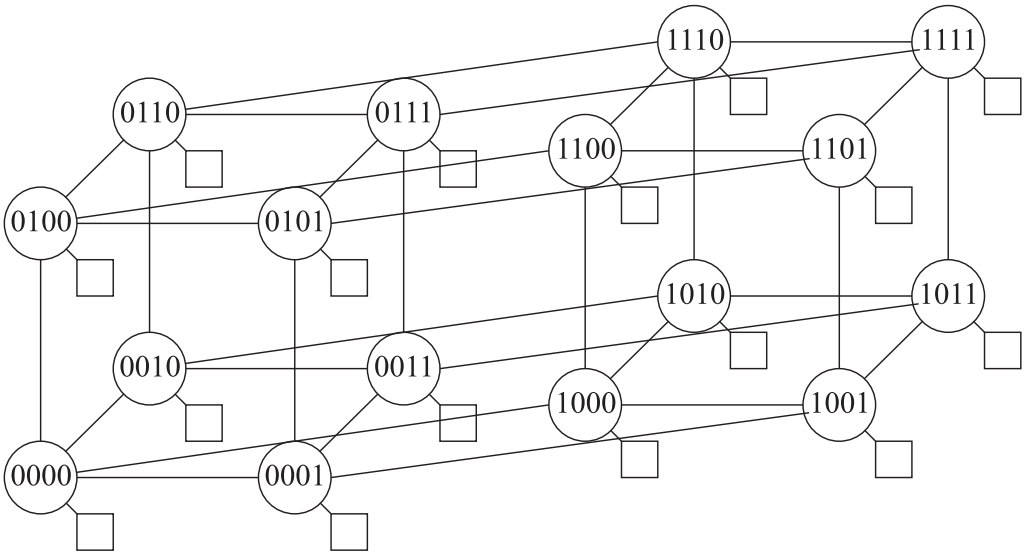
Slika 3.11: Omrežje s topologijo metulja. Stikala v vrstici d so narisana črtkano ker so običajno le sinonim za stikala v vrstici 0.

vozlišče $(0, 5)$, in je povezano s procesorjem 5.

Omrežje s topologijo hiper-kocke Hiper-kocka (*ang.* hypercube), ki jo imenujemo včasih tudi binarna n-kocka, je metulj, v katerem je vsak stolpec stikalnih vozlišč združen v eno samo vozlišče. Binarna n-kocka je sestavljena iz $n = 2^d$ procesnih vozlišč in enakega števila stikalnih vozlišč. Zato je primer neposredne topologije. Procesorska vozlišča in povezana stikalna vozlišča so označena od 0 do $2^d - 1$. Dve stikalni vozlišči sta sosedni, če se njuni binarni oznaki razlikujeta na natanko enem mestu. Slika 3.12 prikazuje 4-dimenzionalno hiper-kocko.

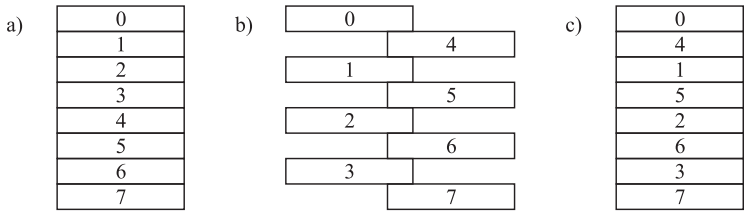
Premer hiper-kocke z $n = 2^d$ stikalnimi vozlišči je enak $\log_2 n$, razpolovna širina pa $n/2$. Majhen premer in visoka razpolovna širina pa pogojujeta druga faktorja. Število povezav na stikalo je $\log_2 n$ brez upoštevanja povezave s procesorjem, najdaljša povezava v hiper-kocki pa narašča z naraščanjem števila vozlišč v omrežju.

Pri pošiljanju sporočil se izkorišča dejstvo, da se binarne oznake sosednjih stikal razlikujejo le na enem mestu. Če želimo, na primer, v 4-dimenzionalni hiper-kocki poslati sporočilo od stikala (procesorja) 0101 do 0011, potem lahko uporabimo dve poti: $0101 \rightarrow 0001 \rightarrow 0011$ ali $0101 \rightarrow 0111 \rightarrow 0011$.



Slika 3.12: Hiperkocka z 16 procesorskimi in 16 stikalnimi vozlišči.

Mešalno omrežje Mešalno omrežje (*ang.* Shuffle Exchange Network) je zasnovano na ideji o idealnemu mešanju. Vzemimo, da urejene karte razdelimo na dve polovici, nato pa izvedemo popolno mešanje, kot je to prikazano na sliki 3.13. Če

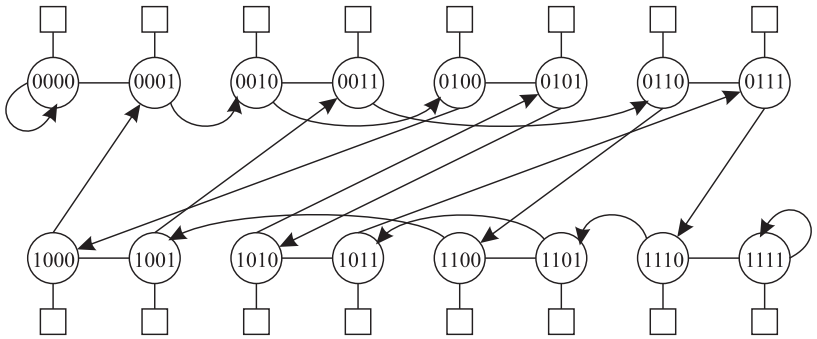


Slika 3.13: Prikaz popolnega mešanja na kartah.

začetni položaj kart predstavimo z binarnim številom, potem lahko nov položaj izračunamo s cikličnim vrtenjem binarnega števila v levo. Na primer, karta 5 ima originalni indeks $5 = 101_2$, po popolnem mešanju pa indeks $3 = 011_2$.

Mešalno omrežje je neposredna topologija z $n = 2^d$ procesorji/stikali, ki so označena binarno s števili od 0 do $n - 1$. Slika 3.14 prikazuje mešalno omrežje s 16 vozlišči. Stikala imajo dva tipa povezav: mešalne in zamenjalne. Zamenjalni tip povezuje pare stikal, katerih oznake se razlikujejo po najmanj pomembnem bitu. Mešalni tip pa povezuje stikalo i s stikalom j, kjer j dobimo s cikličnim vrtenjem binarne predstavitve števila i za eno mesto v levo. Na primer, v omrežju s 16 vozlišči mešalni tip povezuje stikalo $5(0101_2)$ s stikalom $10(1010_2)$.

Vsako stikalo v mešalnem omrežju ima konstantno število povezav: dve izhodni in dve vhodni (povezava s procesorjem ni upoštevana). Dolžina najdaljše povezave se



Slika 3.14: Mešalno omrežje s 16 vozlišči. Mešalne povezave so podane s puščico, zamenjalne pa brez.

povečuje z večanjem omrežja. Premer mešalnega omrežja z n stikali je $2 \cdot \log_2 n - 1$, razpolovna širina pa je približno enaka $n/\log_2 n$.

Oglejmo si še povezovanje (*ang.* routing) pri prenosu sporočila po omrežju. Najdaljša pot gre po $\log_2 n$ zamenjalnih povezavah (Z) in $\log_2 n - 1$ mešalnih povezavah (M). Najslabši scenarij zahteva prenos sporočila od stikala 0 do stikala $n - 1$ (ali obratno). Vzemimo, da je število vozlišč enako $n = 16$. Prenos sporočila od 0000 do 1111 zahteva $2\log_2 n - 1 = 7$ korakov:

$$0000 \xrightarrow{Z} 0001 \xrightarrow{M} 0010 \xrightarrow{Z} 0011 \xrightarrow{M} 0110 \xrightarrow{Z} 0111 \xrightarrow{M} 1110 \xrightarrow{Z} 1111 \quad . \tag{3.1}$$

Ni težko predvideti algoritma, ki vedno generira poti, ki tvorijo $\log_2 n - 1$ mešalnih povezav, toda preskočijo zamenjalne povezave tam, kjer nižjega bita ni potrebno spremeniti. Na primer, prenos sporočila od 0011 do 0101 gre lahko po naslednji poti:

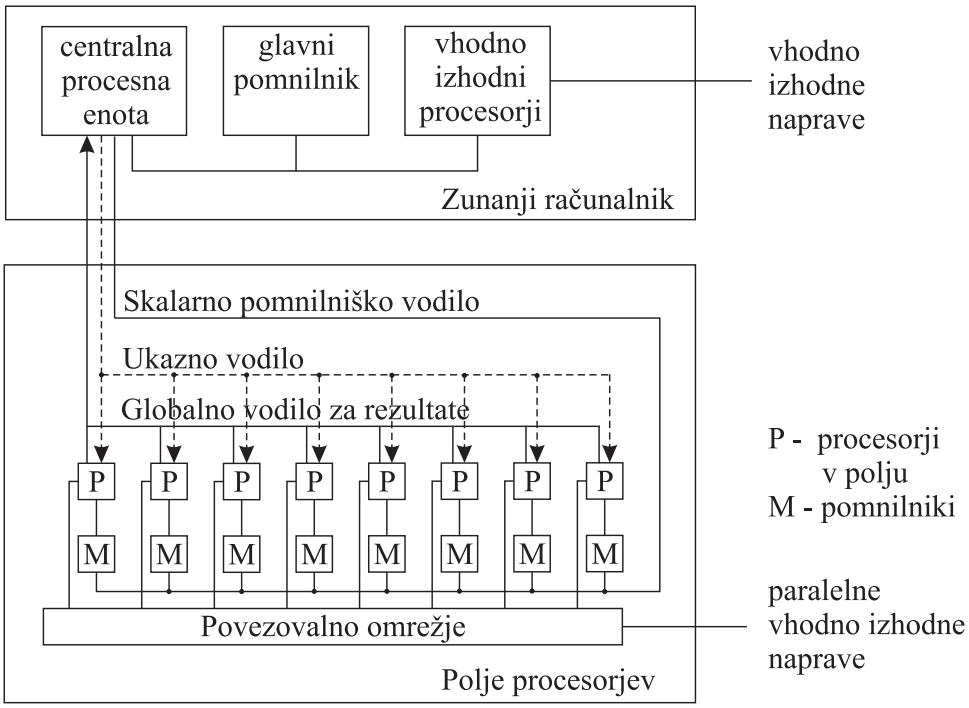
$$0011 \xrightarrow{Z} 0010 \xrightarrow{M} 0100 \xrightarrow{Z} 0101 \xrightarrow{M} 1010 \xrightarrow{M} 0101 \quad . \tag{3.2}$$

Z izboljšanim algoritmom, bi bilo mogoče ugotoviti, da zadnji menjavi nista potrebni.

3.4.2 Polja procesorjev

Vektorski računalnik je računalnik, katerega ukazi izvajajo operacije nad vektorji in skalarji. Obstajata dve možni izvedbi vektorskega računalnika. Prva je cevovodni vektorski procesor, ki prenaša vektorje iz pomnilnika v centralno procesno enoto, kjer jih uporabi cevovodna aritmetična enota. Predstavnika te variante sta

superračunalnika CRAY-1 in CYBER-205. Drugo možnost pa predstavlja polje procesorjev, ki je kombinacija sekvenčnega računalnika in niza identičnih, sinhroniziranih procesnih elementov, ki so sposobni hkrati procesirati iste operacije nad različnimi podatki. Slika 3.15 podaja tipično arhitekturo polja procesorjev.



Slika 3.15: Tipična arhitektura polja procesorjev.

Zunanji (*ang.* front-end) računalnik je pri tem standardni eno-procesorski sekvenčni računalnik. Njegov glavni pomnilnik vsebuje program in podatke, ki se izvajajo sekvenčno. Polje procesorjev je razdeljeno na mnogo parov procesor - pomnilnik. Podatki za paralelno procesiranje so porazdeljeni po lokalnih pomnilnikih v polju procesorjev. Za izvajanje paralelnih operacij, prenese zunanji računalnik ukaz v polje procesorjev, ki ga nato paralelno izvede nad operandi v lokalnih pomnilnikih.

V primeru 1024 procesorjev v polju se ukaz, ki sešteva dva vektorja dolžine N izvede enako hitro za vse $N \leq 1024$. V primeru, da je $N > 1024$ pa je potrebno porazdeliti podatke po lokalnem pomnilniku neenakomerno. Delitev podatkov seveda vpliva na celotni čas računanja, poleg tega pa je celotni čas odvisen še od načina povezovanja procesorjev, kar si bomo podrobneje ogledali kasneje. Na primer, za $N = 10.000$ lahko 784 procesorjev dobi po 10 podatkov, 240 procesorjev pa po 9 podatkov. Tedaj izvajanje seštevanja traja 10 krat dlje kot v primeru $N \leq 1024$.

Seveda je tipično paralelno računanje precej bolj zapleteno kot preprosto seštevanje vektorjev. Pogosto je nova vrednost vektorskega ali matričnega elementa odvisna

od drugih elementov. Na primer, metoda končnih diferenc pri reševanju parcialne diferencialne enačbe zahteva računanje enačbe

$$a_i \leftarrow \frac{1}{2}(a_{i-1} + a_{i+1}) \quad . \tag{3.3}$$

Tedaj potrebuje procesor podatke, ki so v lokalnem pomnilniku drugih procesorjev in ki jih lahko pridobi preko povezovalnega omrežja. Najbolj popularna topologija preklopnega omrežje v poljih procesorjev je dvo-dimenzionalna mreža. Poleg prednosti, ki smo jih spoznali v prejšnjem poglavju, je dodatna kvaliteta tega omrežja, da ga je mogoče implementirati v VLSI.

Polja procesorjev imajo kar nekaj slabosti, ki so povzročila njihov zaton:

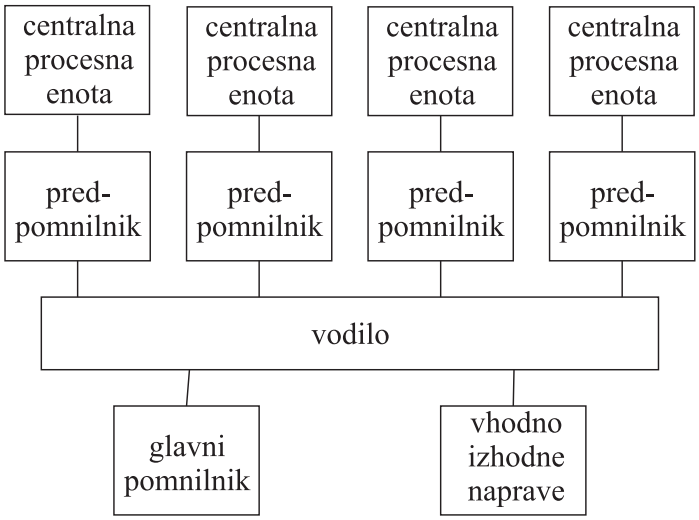
- Veliko problemov ni mogoče obravnavati z vidika podatkovnega paralelizma.
- Ker polje procesorjev v vsakem trenutku izvaja le en ukaz, se njegova učinkovitost precej zmanjša pri tipično ne-paralelnih oziroma pogojnih ukazih, posebno pri vgnezdenih stavkih, na primer, IF-THEN-ELSE.
- Polje procesorjev je tipičen predstavnik enouporabniških paralelnih sistemov. Prehod na večuporabniški sistem, ki bi omogočal hkratno izvajanje več paralelnih programov, je zapleten.
- Dobro skaliranje problemov ni mogoče. Za visoko zmogljivo delovanje je potreben zelo prepusten komunikacijski kanal med zunanjim računalnikom in poljem procesorjev ter tudi med procesorji v polju in med procesorji v polju ter vhodno-izhodnimi napravami. Pri velikem številu procesorjev je cena povezovalnega omrežja glede na celotno ceno sistema majhna, pri majhnem številu procesorjev pa je njen delež precej večji.
- Proizvajalci polj procesorjev v tehnologiji VLSI vedno težje tekmujejo s s proizvajalci standardnih splošno namenskih procesorjev.

3.4.3 Večprocesorski sistemi

Večprocesorski računalniški sistem je sistem z več centralnimi procesnimi enotami, ki si delijo skupen glavni pomnilnik. Večračunalniški sistemi rešujejo tri probleme, ki so značilni za polja procesorjev: zgrajeni so lahko iz splošno namenskih procesorjev, naravno podpirajo večuporabniško delovanje, poleg tega pa se jim učinkovitost pri izvajanju tipično zaporednih ukazov ne zmanjša.

Centralizirani večprocesorski sistemi

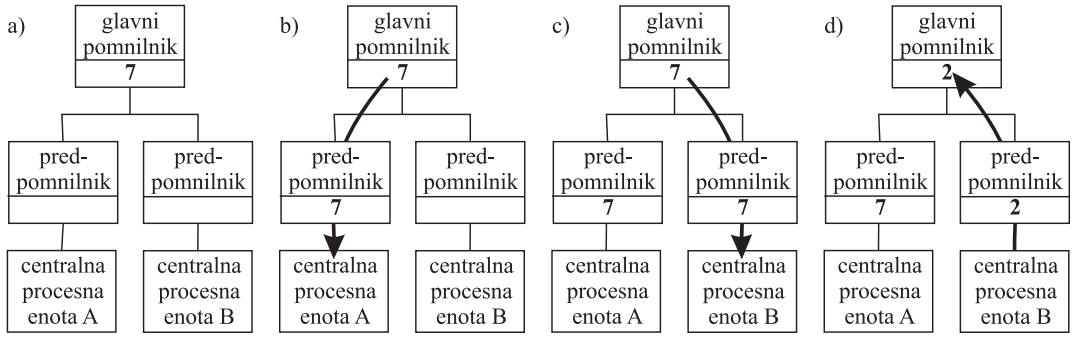
Pri centraliziranih večprocesorskih sistemih je glavni pomnilnik v enem delu oziroma je centraliziran. Centralizirani večprocesorski sistemi so tako najbolj naravna razširitev iz enoprocesorskih sistemov. Dodatne procesne enote so povezane z vodilom tako, da si vsi procesorji delijo isti glavni pomnilnik (slika 3.16). Tej arhitekturi pravimo večprocesorski sistem z enotnim dostopom do pomnilnika (*ang.* Uniform Memory Access - UMA) ali tudi simetrični večprocesorski sistem (*ang.* Symmetric Multiprocessor - SMP). Za te sisteme je značilno, da je dostopni čas do glavnega pomnilnika za vse procesorje enak. Kljub temu, da imajo procesne enote lokalne predpomnilnike, ki zmanjšajo obremenitev vodila in glavnega pomnilnika, prepustnost vodila omejuje število procesnih enot na nekaj deset.



Slika 3.16: Arhitektura centraliziranega večprocesorskega računalniškega sistema.

Privatni podatki so tisti, ki jih uporablja le en sam procesor, medtem ko so deljeni podatki tisti, ki jih lahko uporabljajo vsi procesorji. Procesorji komunicirajo preko deljenih podatkov. Pri snovanju sistemov je zato potrebno posebno pozornost nameniti problemoma skladnosti (koherentnosti) predpomnilnikov in glavnega pomnilnika ter sinhronizacije.

Če ne bi bilo ustreznega osveževanja predpomnilnikov in glavnega pomnilnika, bi lahko prišlo do stanja, v katerem bi imeli različni procesorji različne vrednosti za iste pomnilniške naslove. Problem je ilustriran na sliki 3.17. Najbolj pogosto se problem skladnosti predpomnilnika in glavnega pomnilnika rešuje z zagotavljanjem ekskluzivnega predpomnilniškega dostopa procesorja do glavnega pomnilnika pred spremembo podatka v glavnem pomnilniku. To se izvede z razveljavitvijo vseh predpomnilnikov blokov drugih procesorjev, ki hranijo isti podatek. Tedaj lahko procesor spremeni podatek v svojem predpomnilniku in tudi v glavnem pomnilniku.



Slika 3.17: Ilustracija problema skladnosti predpomnilnika in glavnega pomnilnika.

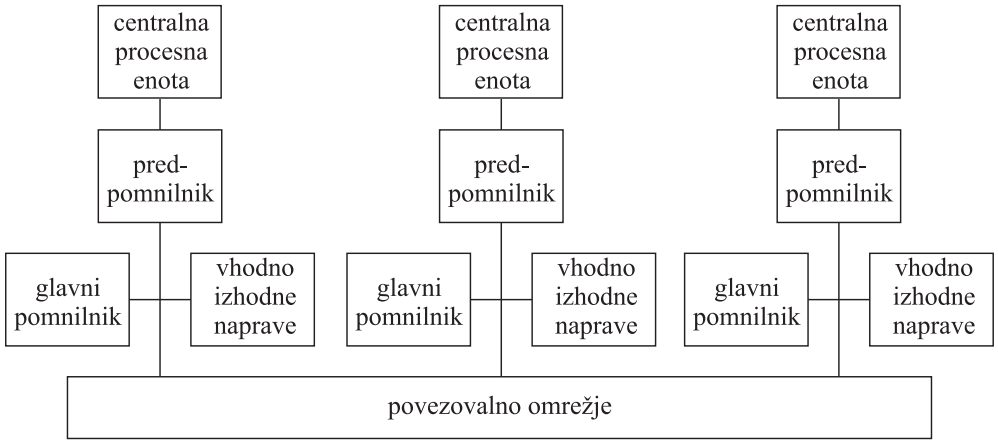
Če v istem času želi drug procesor uporabiti isti podatek v svojem predpomnilniku, dobi informacijo, da podatek ni veljaven in da ga je potrebno obnoviti. Ta postopek se imenuje zapis z razveljavitvijo (*ang.* write invalidate protocol). Če želita dva procesorja hkrati vpisovati na isto pomnilniški naslov, zmaga le eden, drugemu pa se predpomnilniški blok razveljavi in da si mora pridobiti novo kopijo podatkov iz glavnega pomnilnika.

Pri komunikaciji med procesorji je potrebnih več oblik sinhronizacije. Ena od njih je medsebojna izključitev (*ang.* mutual exclusion), pri kateri je naenkrat lahko aktiven največ en procesor. Sinhronizacija z zapornicami (*ang.* barrier synchronization) pa zagotavlja, da se procesi v programu ne bodo izvajali od točke imenovane zapornica naprej, dokler te točke ne dosežejo vsi procesi. Ta oblika sinhronizacije običajno uporablja sinhronizacijske ukaze podprte v strojni opremi, kot so na primer ukaz, ki se ga ne da prekiniti, ali ukaz, ki pridobi in spremeni stanje v enem koraku.

Porazdeljeni večprocesorski sistemi

S porazdelitvijo glavnega pomnilnika med procesorje dobimo porazdeljeni večprocesorski računalniški sistem, pri katerem je dostop do lokalnega pomnilnika veliko hitrejši kot do ostalega dela glavnega pomnilnika, oziroma do lokalnih pomnilnikov drugih procesorjev. Arhitekturo takega sistema prikazuje slika 3.18. Zaradi prostorske in časovne komponente, ki vplivata na izvajanje programa, je mogoče porazdeliti ukaze in podatke po pomnilniških modulih tako, da je večina pomnilniških klicev vezanih na lokalni pomnilnik, kar povečuje pomnilniško prepustnost in s tem hitrost procesiranja.

Če porazdeljeni pomnilnik tvori en naslovni prostor, potem govorimo o porazdeljenem večprocesorskem sistemu. Tedaj z enakim naslovom na različnih procesorjih naslavljamo isto pomnilniško lokacijo. Takšnim sistemom pravimo tudi večprocesorski sistem z ne-enoličnim dostopom do pomnilnika (*ang.* Non-Uniform Memory Access - NUMA), ker je čas dostopa odvisen od tega, ali je naslovljen lokalni pomnilnik ali



Slika 3.18: Arhitektura porazdeljenega večprocesorskega računalniškega sistema.

ne.

Pri porazdeljenih večprocesorskih sistemih je potrebno uporabiti drugačno rešitev za problem skladnosti predpomnilnikov in glavnega pomnilnika. Zelo uporabljan je protokol direktorija (*ang.* directory-based protocol), kjer direktorij vsebuje podatke o delitvi pomnilniških modulov med predpomnilnike procesorjev na naslednji način:

- Modul ni v predpomnilniku (*ang.* non-cached)
- Modul je deljen (*ang.* shared), kar pomeni, da je predpomnjen v več procesorjih in da so kopije pravilne
- Modul je ekskluziven (*ang.* exclusive), kar pomeni, da je predpomnjen pri enem samem procesorju in da je kopija v pomnilniku nepotrebna.

Direktorij je porazdeljen, da ne bi predstavljal ozkega grla, vendar pa njegova vsebina ni replicirana, kar pomeni, da je podatek o vsakem pomnilniškem modulu le na enem mestu.

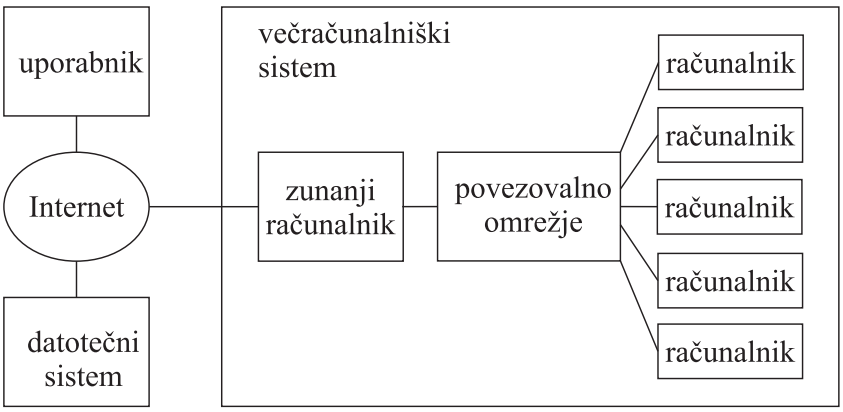
3.4.4 Večračunalniški sistemi

Večračunalniški sistemi so še en primer računalniških sistemov s porazdeljenim pomnilnikom in s številnimi procesnimi enotami. Za razliko od večprocesorskih sistemov tipa NUMA, ki imajo eno globalno naslovno področje, imajo večračunalniški sistemi neodvisna lokalna naslovna področja. Vsak procesor ima dostop samo do svojega lokalnega pomnilnika. Z istim naslovom na različnih procesorjih naslavljamo različne pomnilniške lokacije na fizično različnih pomnilnikih. Ker ni deljenega naslovnega prostora, procesorji komunicirajo s prenašanjem sporočil.

Večračunalniški sistemi lahko uporabljajo specifično povezovalno omrežje, da dosežejo nizko stopnjo latence in visoko prepustnost oziroma dostopnost med procesorji. Po drugi strani pa univerzalni računalniki, ki so na voljo na trgu, povezani s standardnimi stikali, tvorijo lokalno omrežje (*ang.* commodity clusters), ki je bistveno cenejše, vendar pa z višjo stopno latence in nižjo prepustnostjo.

Nesimetrični večračunalniški sistemi

Prvi večračunalniški sistemi so imeli nesimetrično arhitekturo, ki je prikazana na sliki 3.19. Zunanji računalnik (*ang.* front-end) je namenjen za komunikacijo z upo-



Slika 3.19: Arhitektura nesimetričnega večračunalniškega sistema.

rabniki in vhodno-izhodnimi napravami, notranji računalniki (*ang.* back-ends) pa so namenjeni za obdelovanje podatkov. Primera nesimitričnih večračunalniških sistemov sta Intel iPSC in nCUBE/ten. Prvi ima 128 procesorjev, drugi pa 1024.

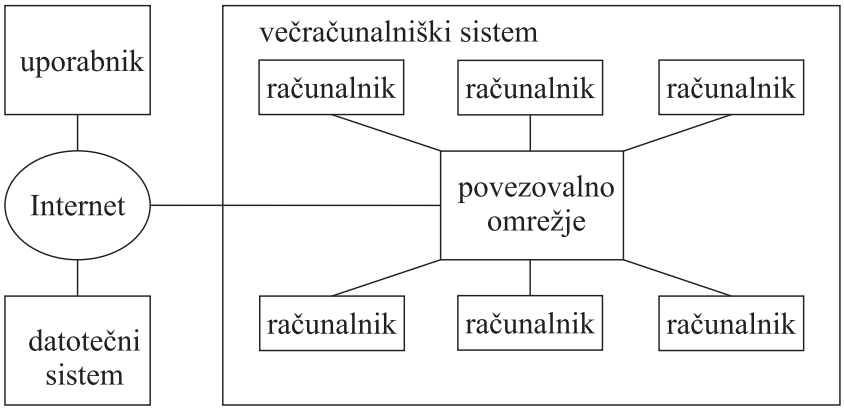
Nesimetrični večračunalniški sistem ima nekaj izrazitih pomanjkljivosti.

- Prvič, če odpove zunanji računalnik, odpove celoten večračunalniški sistem, ker ni več mogoča komunikacija z zunanjim svetom.
- Drugič, razširljivost (skalabilnost) je odvisna od lastnosti zunanjega računalnika, saj se uporabniki preko njega prijavljajo v sistem in na na njem urejajo in prevajajo programe.
- Naslednjo hibo predstavlja razhroščevanje, ki je zaradi primitivnih operacijskih sistemov v notranjih računalnikih zelo oteženo.
- In končno, vsak paralelni program zahteva razvoj dveh različnih programov: enega za zunanji računalnik in drugega za notranje računalnike. Prvi je potreben za interakcijo z uporabnikom in datotečnim sistemom, za pošiljanje

podatkov k notranjim računalnikom in za prenos rezultatov v zunanji svet. Drugi je potreben samo za računski del algoritma, ki se izvaja na notranjih računalnikih.

Simetrični večračunalniški sistemi

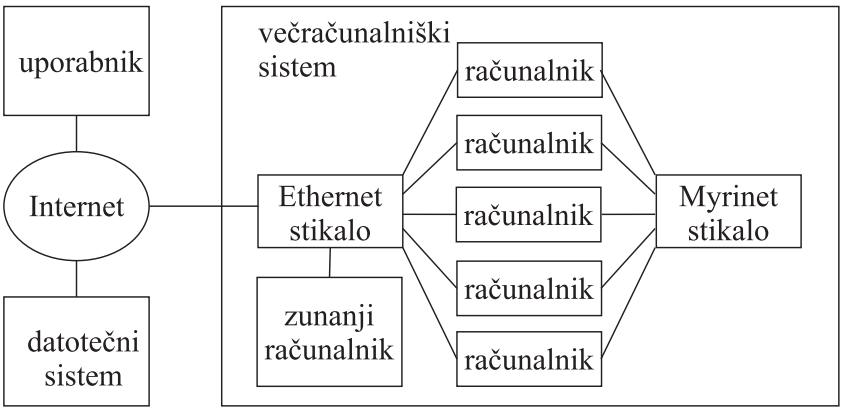
Pri simetričnih večračunalniških sistemih vsak računalnik uporablja isti operacijski sistem in ima potencialno enako funkcionalnost. Uporabniki se lahko prijavljajo v sistem na vsakem računalniku, prav tako na vsakem računalniku lahko urejajo programe in jih prevajajo. Vsi računalniki so lahko tudi vključeni v izvajanje paralelnih programov. Arhitekura simetričnega večračunalniškega sistema je prikazana na sliki 3.20.



Slika 3.20: Arhitektura simetričnega večračunalniškega sistema.

Simetrični večračunalniški sistem odpravlja številne težave, ki smo jih našteli pri nesimetričnem večračunalniškem sistemu, na primer odpravlja ozko grlo zunanjega računalnika, izboljša postopek razhroščevanja, ... Slabosti simetričnega večračunalniškega sistema pa so predvsem v zahtevnem razporejanju obremenitev procesorjev oziroma v težko dosegljivi visoki zmogljivosti sistema.

Najboljša rešitev je mešanica med simetričnim in nesimetričnim večračunalniškim sistemom, kjer so zunanji in vsi notranji računalniki povezani preko Ethernet omrežja, notranji računalniki pa so poleg tega povezani še na omrežje, ki omogoča hiter dostop do podatkov (slika 3.21).



Slika 3.21: Mešanica simetričnega in nesimetričnega modela večračunalniškega sistema.

Razlike med omrežjem računalnikov in gručo računalnikov

Omrežje računalnikov običajno tvori niz oddaljenih namiznih računalnikov (delovnih postaj), povezanih s povezovalnim medijem Ethernet. Osnovni namen namiznih računalnikov je, da služijo uporabniku. Izvajanje paralelnih programov pri tem le zapolni proste procesorske cikle. Posamezni računalniki imajo lahko različne operacijske sisteme, na njih se lahko izvajajo različni programi. Uporabniki lahko svoje računalnike izklopijo.

Gruča računalnikov je običajno niz računalnikov (iz masovne proizvodnje), ki so nameščeni na istem prostoru in so namenjeni za izvajanje paralelnih programov. Posamezni računalniki so dostopni le preko stikalnega omrežja in tipično nimajo svojih zaslonov in tipkovnic. Vsi imajo isto verzijo enakega operacijskega sistema in imajo identične slike lokalnih diskov. Gruča je administrirana kot ena entiteta.

Pomembna razlika med omrežji in gručami je v hitrosti prenosa podatkov. Ethernet je prepočasen za gručo, poleg tega je pomembno, da je omrežje stikalno in ne deljeno. Najbolj popularna naslednja omrežja za gruče so navedena v tabeli 3.1.

Tabela 3.1: Primerjava prenosnih medijev.

	Latenca [μs]	Prepustnost [Mbit/s]
Hitri Ethernet	100	100
Gigabit Ethernet	100	1000
Myrinet	7	1920
10 Gb Ethernet	2,4	9600

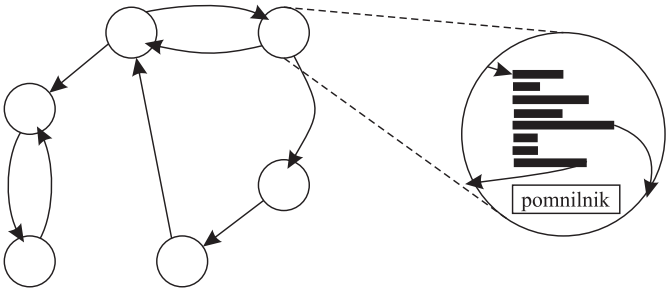
Poglavje 4

Snovanje paralelnih algoritmov

4.1 Model opravilo/kanal

Razvoj paralelnih programov se razlikuje od razvoja klasičnih zaporednih programov. Zahteva posebno vrsto razmišljanja, ki upošteva vse značilnosti ciljnih paralelnih programov. Ian Foster je v pomoč pri razvoju paralelnih programov predlagal postopek, zasnovan na modelu opravilo/kanal (*ang.* task/channel) [18], ki vključuje vse bistvene korake pri razvoju paralelnih programov. Njegov postopek olajša razvoj učinkovitih paralelnih programov, posebno tistih, ki se izvajajo na paralelnih računalnikih s porazdeljenim pomnilnikom.

Model opravilo/kanal predstavlja paralelno računanje z nizom opravil, ki med seboj komunicirajo tako, da si pošiljajo sporočila po kanalih. Osnovna ideja je prikazana na sliki 4.1. Pod opravilom si predstavljamo program, lokalni pomnilnik in vho-



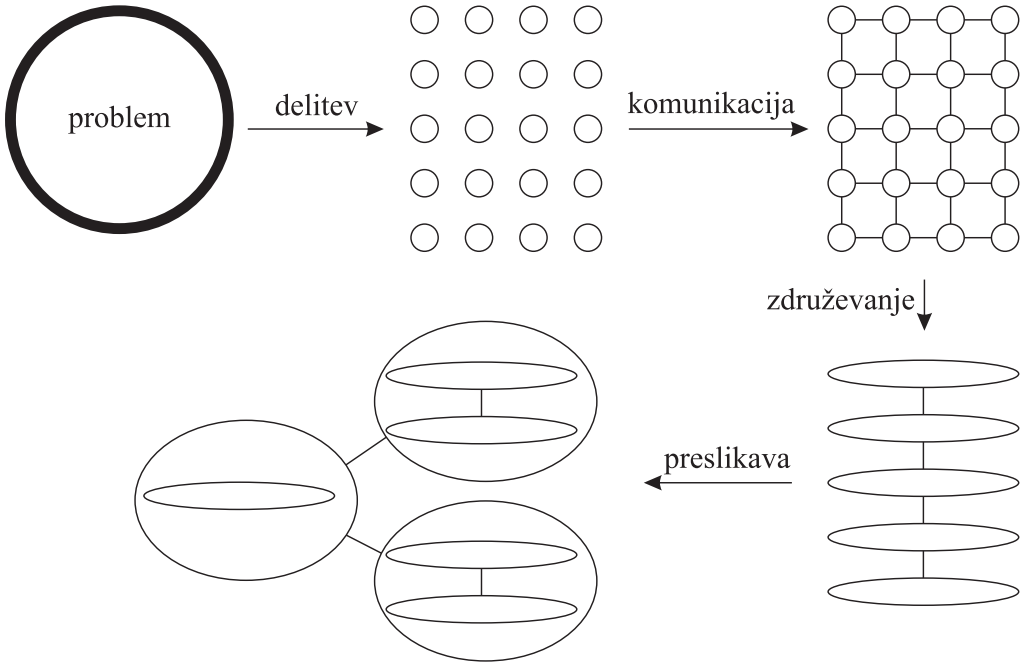
Slika 4.1: Model opravilo/kanal.

dno/izhodna vrata. Lokalni pomnilnik vsebuje ukaze programa in podatke. Opravilo lahko pošlje svoje lokalne podatke drugim opravilom preko izhodnih vrat. Kanal je sporočilna vrsta (*ang.* message queue), ki poveže izhodna vrata enega opravila z vhodnimi vrati drugega. Podatki na vhodu si sledijo v enakem vrstnem redu, kot

so bili poslani na izhodni strani. Opravilo ne more sprejeti podatka, dokler ta ni bil poslan s strani drugega opravila. Sprejem je blokiran, če na vhodu ni podatka. Po drugi strani pa oddajanje ni nikoli blokirano, tudi če prejšnje sporočilo še ni bilo sprejeto. Zato je sprejem sinhrona operacija, oddajanje pa asinhrona operacija.

4.2 Fosterjeva metodologija

Foster je predlagal postopek snovanja paralelnih algoritmov, ki ga sestavljajo štirje koraki: delitev (*ang.* partitioning), komunikacija (*ang.* communication), združevanje (*ang.* agglomeration) in preslikava (*ang.* mapping). Postopek je shematsko prikazan na sliki 4.2.



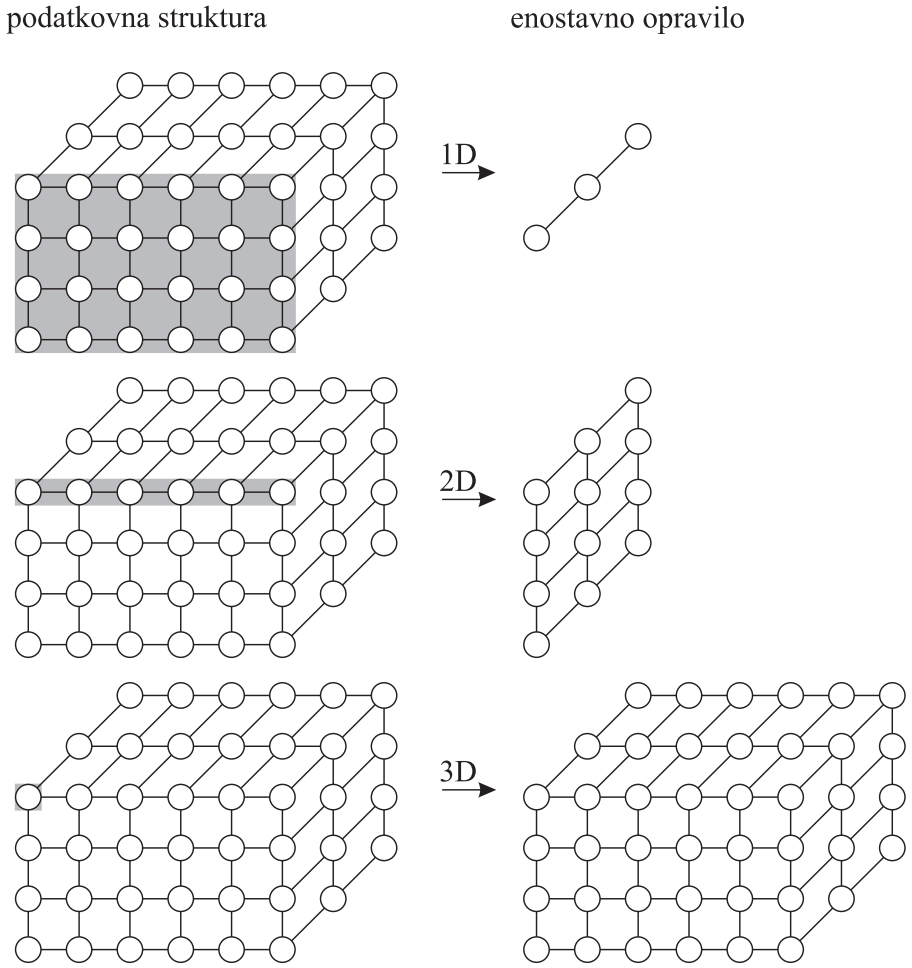
Slika 4.2: Fosterjeva metodologija snovanja paralelnih algoritmov.

4.2.1 Delitev

Delitev je proces, s katerim razdelimo podatke in računske operacije na majhne dele. Kadar delimo podatke, govorimo o podatkovni delitvi, v primeru da delimo računske operacije, pa o funkcijski delitvi.

Pri podatkovni delitvi najprej delimo podatke na majhne dele, nato pa šele določimo,

kako jih povežemo s potrebnimi računskimi operacijami. Vedno se je dobro osredotočiti na največje in najpogosteje naslavljane podatkovne strukture v programu. Primer različnih načinov podatkovne delitve, ki vplivajo na različno zasnovane računske postopke, prikazuje slika 4.3.



Slika 4.3: Primeri podatkovne delitve.

Funkcijska delitev je komplementarni postopek, po katerem najprej delimo računske operacije na majhne dele in nato določimo, kako te dele povezati s podatki. Pogosto na ta način pridemo do zbirke opravil, za katere je značilen cevovodni paralelizem. Naloga funkcijske delitve je identificirati čim več osnovnih opravil, saj je njihovo število zgornja meja možnega paralelizma v algoritmu.

Lastnosti, ki določajo kvaliteto procesa delitve, so

- število identificiranih osnovnih opravil je za velikostni razred večje od števila procesorjev v paralelnem računalniku,

- kompleksnost osnovnih opravil je približno enaka,
- število redundantnih podatkov in redundantnih računskih operacij je minimalno in
- število osnovnih opravil narašča z naraščanjem velikosti problema.

4.2.2 Komunikacija

Ta korak določa komunikacijski vzorec med osnovnimi opravili. Vzorec je lahko lokalni ali globalni. Kadar opravila potrebujejo podatke le od majhnega števila drugih opravil, je vzorec komunikacije lokalni, v primeru odvisnosti od velikega števila procesov pa je vzorec globalni. Podrobnejša specifikacija komunikacije na tej stopnji razvoja paralelnega algoritma ni potrebna.

Komunikacija med opravili (procesi) predstavlja dodaten strošek (*ang.* overhead) paralelnega algoritma, saj je to nekaj, česar zaporedni algoritem ne potrebuje. Zato je minimizacija komunikacije pomemben cilj v razvoju paralelnega algoritma.

Lastnosti kvalitetne komunikacije so:

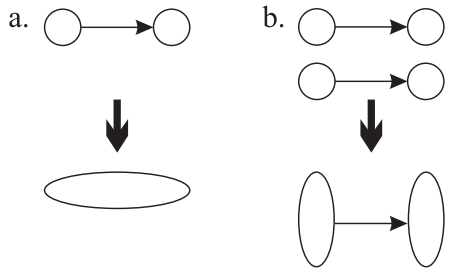
- komunikacijske operacije morajo biti uravnotežene med opravili,
- vsako opravilo komunicira le z majhnim številom sosedov,
- opravila lahko komunicirajo sočasno in
- opravila lahko izvajajo računanje sočasno.

4.2.3 Združevanje

Osnovna opravila je potrebno združevati v večja sestavljena opravila zato, da se izboljšajo lastnosti paralelnega algoritma in da se poenostavi samo programiranje. Pogosto opravila združujemo tako, da enemu procesorju pripada eno sestavljeno opravilo. Tedaj je naslednji korak Fosterjevega postopka, to je preslikava, trivialen.

Prvi cilj združevanja je zmanjšati strošek komunikacije. Če združujemo osnovna opravila, ki komunicirajo med seboj, potem smo to komunikacijo povsem odpravili. Tudi opravila, ki si sledijo zaporedno in si pri tem predajajo podatke, je smiselno združevati iz istega razloga. Zmanjševanje stroškov komunikacije dosežemo tudi z združevanjem opravil, ki pošiljajo oziroma oddajajo sporočila (slika 4.4). Pošiljanje manjšega števila daljših sporočil vzame manj časa kot pošiljanje večjega števila

krajših sporočil. To je posledica latence ob začetku prenosa, ki je neodvisna od dolžine sporočila.



Slika 4.4: Dva primera združevanja opravil.

Drugi cilj združevanja je ohraniti skalabilnost pri razvoju paralelnega algoritma. To pomeni, da je mogoč prenos na večji paralelni računalnik z več procesorji brez opaznejše spremembe programa.

Tretji cilj združevanja pa je zmanjšanje časa programiranja. Predelava zaporednega programa v ustrezno paralelno različico je ob primerni izvedbi postopka združevanja lahko zelo enostavna.

Lastnosti kvalitetno opravljenega postopka združevanja opravil so:

- povečana lokalnost paralelnega algoritma,
- čas, potreben za redundantno računanje, je krajši od časa, potrebnega za nadomestno komunikacijo,
- obseg redundatnih podatkov je dovolj majhen, da omogoča dobro skaliranje algoritma,
- računski in komunikacijski stroški sestavljenih opravil so podobni,
- število sestavljenih opravil narašča z velikostjo problema,
- število sestavljenih opravil je čim manjše, pa vendar vsaj tako veliko kot je število procesorjev in
- združevanje mora opravičiti ceno spreminjanja zaporedne programske kode v paralelno.

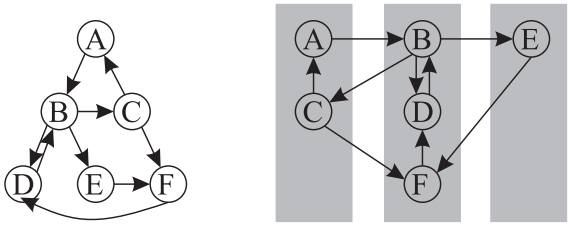
4.2.4 Preslikava

Preslikava je proces dodeljevanja opravil procesorjem. Če izvajamo program na centraliziranem večprocesorskem sistemu, potem operacijski sistem avtomatično dodeli

opravila procesorjem. Pri večračunalniških sistemih s porazdeljenim pomnilnikom pa ta korak zahteva dodatno pozornost.

Cilj preslikave je povečati izkoriščenost procesorjev in hkrati zmanjšati medprocesorsko komunikacijo, kar sta običajno nasprotujoči zahtevi. Izkoriščenost procesorjev je določena s povprečnim deležem časa, v katerem procesorji aktivno izvajajo opravila, potrebna za rešitev problema. Procesorji so maksimalno izkoriščeni takrat, ko vsi istočasno pričnejo procesirati in vsi istočasno tudi končajo delo. Neaktivnost kateregakoli procesorja zmanjšuje izkoriščenost procesorjev.

Komunikacija med procesorji naraste, ko se dve opravili, povezani s kanalom, preslikata na dva različna procesorja. Analogno se komunikacija med procesorji zmanjša, če se dve opravili, povezani s kanalom, preslikata v en procesor (slika 4.5).



Slika 4.5: Primer preslikave.

Na primer, če imamo p procesorjev in, če preslikamo vsa opravila na en procesor, smo zmanjšali medprocesorsko komunikacijo na 0, vendar hkrati zmanjšali izkoriščenost procesorjev na $1/p$. Pravi cilj je poiskati ustrezno vmesno točko med maksimalno izkoriščenostjo procesorjev in minimalno komunikacijo. Žal je ta problem NP-poln, kar pomeni, da ni znanega algoritma s polinomsko časovno kompleksnostjo, ki bi preslikal opravila k procesorjem tako, da bi bil čas izvajanja minimalen. Zato nam ostanejo hevristična pravila, ki lahko dovolj dobro opravijo korak preslikave. Strategija je odvisna od značilnosti sestavljenih opravil, ki so rezultat korakov delitve, komunikacije in združevanja.

4.3 Primer: operacija redukcije

Podan je niz n števil $a_0, a_1, \ldots, a_{n-1}$ in asociativni binarni operator redukcije $\diamond$, s katerim želimo izračunati vrednost izraza $a_0 \diamond a_1 \diamond \ldots \diamond a_{n-1}$. Primeri operacije redukcije so seštevanje, množenje, iskanje največje in najmanjše vrednosti med podanimi števili. V nadaljevanju bomo analizirali problem računanja operacije redukcije nad nizom n števil in pri tem iskali najhitrejšo rešitev s paralelnim programom. Brez izgube na splošnosti in zaradi enostavnejše razlage bomo namesto operatorja redukcije $\diamond$ predpostavljali navadno seštevanje (+). Snovanje paralelnega algoritma

temelji na Fosterjevi metodologiji.

4.3.1 Delitev

Najprej n podatkov razdelimo na najmanjše dele, torej na n kosov. Če vežemo eno opravilo na en podatkovni kos, dobimo n osnovnih opravil, vsako z enim podatkom.

4.3.2 Komunikacija

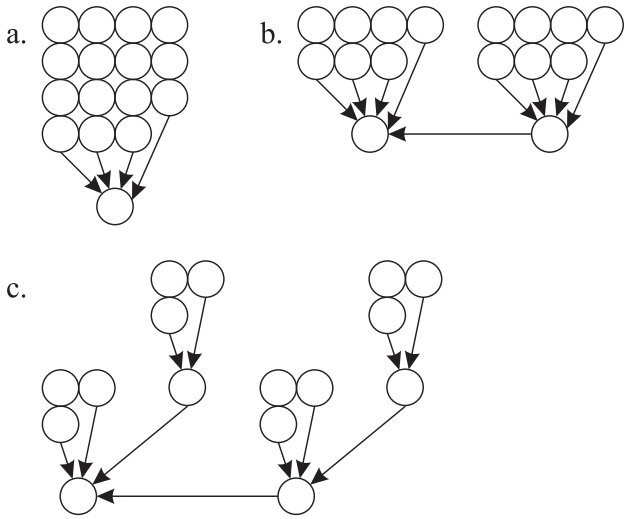
Osnovno opravilo ne more neposredno doseči podatka, ki je v pomnilniku drugega opravila. Zato moramo med opravili vzpostaviti komunikacijske kanale. Kanal med opraviloma A in B dovoljuje opravilu B, da izračuna operacijo seštevanja med obema opraviloma. Želimo, da se komunikacija in računanje izvedeta čim hitreje. Pri tem moramo upoštevati, da v enem komunikacijskem koraku vsako opravilo lahko izvede ali oddajanje ali sprejemanje enega sporočila. Na koncu računanja želimo, da se rezultat nahaja v enem od opravil, imenujemo ga korensko opravilo.

Začnimo z najenostavnejšim pristopom, v katerem vsa opravila pošljejo svoj podatek korenskemu, ki izvede končno operacijo (slika 4.6a). Če komunikacija za prenos podatka zahteva λ časa in izračun vsote χ časa, potem smo s tem pristopom prišli do rešitve, ki traja $(n-1)(\lambda+\chi)$ časa. Pri tem komunikacija vzame $(n-1)\lambda$ časa zaradi sprejema $(n-1)$ podatkov, izračun pa $(n-1)\chi$ časa. Ta rešitev je očitno počasnejša od zaporedne na enem procesorju, ki traja $(n-1)\chi$ časa. Zato moramo najti kompromisno rešitev med komunikacijo in računanjem.

Vzemimo, da imamo dve kvazi-korenski opravili, od katerih je vsako odgovorno za združevanje $n/2$ podatkov (slika 4.6b). Sedaj se dva prenosa podatkov lahko izvedeta paralelno, nato sledita dva paralelno izvedena izračuna. Vsako kvazi-korensko opravilo v času $(n/2-1)(\lambda+\chi)$ določi polovični rezultat. Potrebujemo še en korak komunikacije in en korak računanja, da dobimo končni rezultat v času $(n/2)(\lambda+\chi)$.

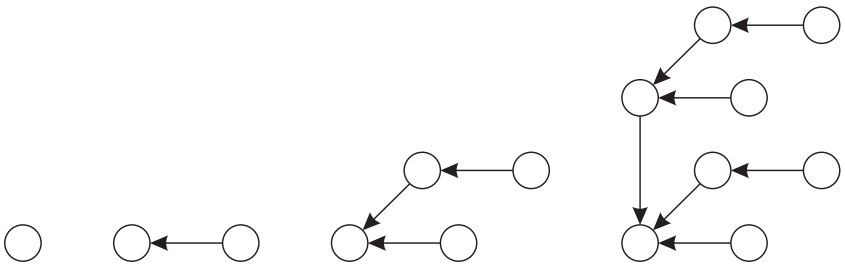
Če nadaljujemo v tej smeri in zopet podvojimo število kvazi-korenov na štiri, potrebujemo po analogiji $(n/4-1)(\lambda+\chi)+2(\lambda+\chi)$ časa, kar je skoraj štirikrat hitrejše od začetnega pristopa (slika 4.6c).

S ponavljanjem zgornjega postopka pridemo do limite, ko imamo $n/2$ kvazi-korenskih opravil, od katerih je vsako odgovorna le za dva podatka. Tedaj se v prvem koraku prenesejo podatki od ene polovice opravil do druge, nato pa opravila, ki so sprejela podatke, izvedejo operacijo redukcije. Vsak korak prenosa podatkov zmanjša število vozlišč na polovico. Pri n podatkih tako za izvedbo redukcije potrebujemo $\log_2 n$ prenosov podatkov. Opisana shema računanja operacije redukcije poteka ustreza



Slika 4.6: Razvoj paralelnega algoritma za izvedbo operacije redukcije za $n = 16$ opravil.

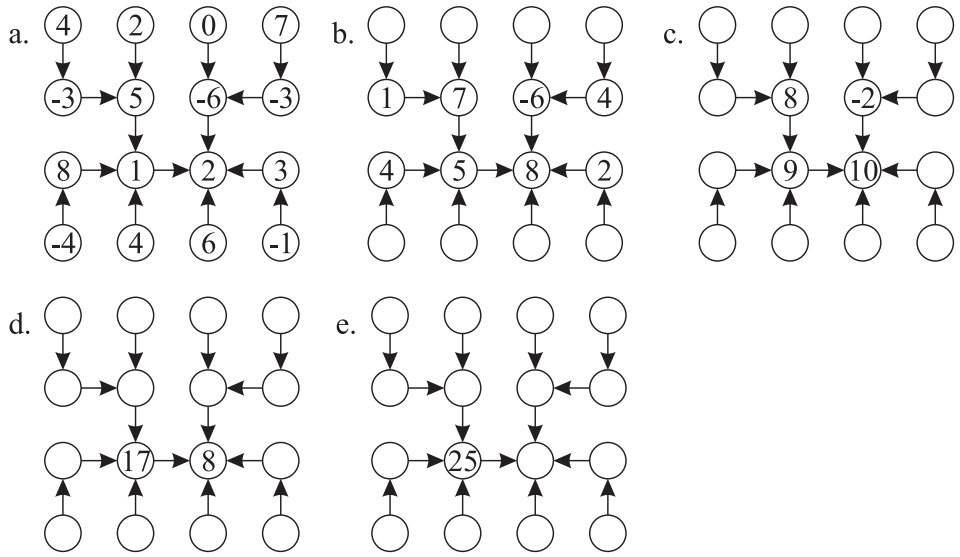
binarnemu drevesu. Binarno drevo, ki je grafično predstavljeno na sliki 4.7, predstavlja eno najpogosteje uporabljanih komunikacijskih topologij pri snovanju paralelnih algoritmov. Pri binarnem drevesu z $n = 2^k$ vozlišči, je največja razdalja od katerega koli vozlišča do korena enaka $k = \log_2 n$.



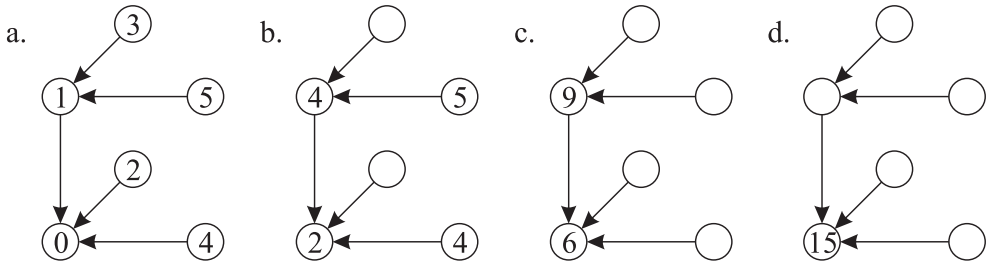
Slika 4.7: Binarna drevesa z 1, 2, 4 in 8 vozlišči.

Na sliki 4.8 je detajlno prikazano, kako je mogoče s kanali, ki imajo topologijo binarnega drevesa, 16 opravil združiti v štirih komunikacijskih korakih.

V primeru, da število vozlišč ni potenca števila 2, moramo spremeniti prvi korak algoritma. Vzemimo, da je $n = 2^k + r$, kjer je $r < 2^k$. Tedaj v prvem koraku izvedemo prenos r podatkov, v nadaljevanju pa si 2^k opravil izmenja podatke enako kot prej. Primer izvajanja operacije redukcije na 6 podatkih prikazuje slika 4.9.



Slika 4.8: Ilustracija komunikacije po kanalih, ki imajo topologijo binarnega drevesa na problemu operatorja redukcije.



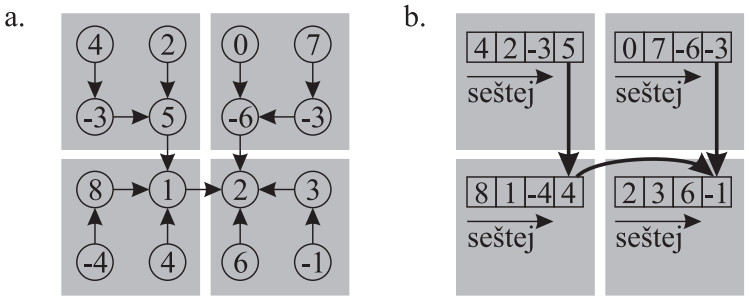
Slika 4.9: Izračun operacije redukcije, ko število opravil $n = 6$ ni potenca števila 2.

4.3.3 Združevanje in preslikava

Pred implementacijo algoritma moramo opraviti še postopka združevanja in preslikave. Slika 4.8a prikazuje shemo opravilo/kanal za paralelni izračun operacije redukcije na $n = 16$ podatkih.

Vzemimo, da imamo p procesorjev, $p \ll n$. Z združevanjem želimo minimizirati komunikacijski čas. To naredimo tako, da združimo n/p osnovnih opravil v okviru vsakega od p procesorjev (slika 4.10a). Namesto n/p osnovnih opravil s po enim podatkom, imamo v okviru vsakega procesorja po eno sestavljeno opravilo z n/p podatki (slika 4.10b).

Ob koncu lahko izpeljemo še izraz za pričakovan čas izvajanja paralelnega algoritma za operacijo redukcije. Vzemimo, da je χ čas, potreben za izvedbo ene operacije seštevanja, in λ čas, potreben za prenos podatka med dvema procesoma.



Slika 4.10: Združevanje in preslikava pri operaciji redukcije $n = 16$ podatkov na $p = 4$ procesorjih.

Če n podatkov razdelimo med p opravil, potem ni nobeno odgovorno za več kot $\lceil n/p \rceil$ podatkov, kjer oklepaja $\lceil\ \rceil$ označujeta zaokrožitev navzgor. Vsak procesor zato izvede računanje v času $(\lceil n/p \rceil - 1)\chi$. Vsaka operacija redukcije nad p opravili je možna v $\lceil \log_2 p \rceil$ korakih, pri čemer se vsak korak z upoštevanjem komunikacije in računanja izvaja $(\lambda + \chi)$ časa. Celotni čas izvajanja programa je tako

$$t_{\text{exe}} = (\lceil n/p \rceil - 1)\chi + \lceil \log_2 p \rceil (\lambda + \chi) \quad . \tag{4.1}$$

Z odvajanjem t_{exe} po p in izenačitvi odvoda z 0, dobimo število procesorjev $p_{\min}$, pri katerem je ocenjeni čas izvajanja programa t_{exe} minimalen. Za zgornji primer je:

$$\begin{aligned} \frac{dt_{\text{exe}}}{dp} &= -\frac{n}{p^2}\chi + \frac{\lambda + \chi}{p} = 0 \\ (\lambda + \chi) &= \frac{n}{p}\chi \\ p_{\min} &= \frac{\chi}{\lambda + \chi} n \quad . \end{aligned} \tag{4.2}$$

Na primer, za $n = 16$ in $\lambda = \chi = 1$, dobimo $p_{\min} = 8$ in $t_{\min} = 7$.

4.4 Primer: delci v gravitacijskem polju

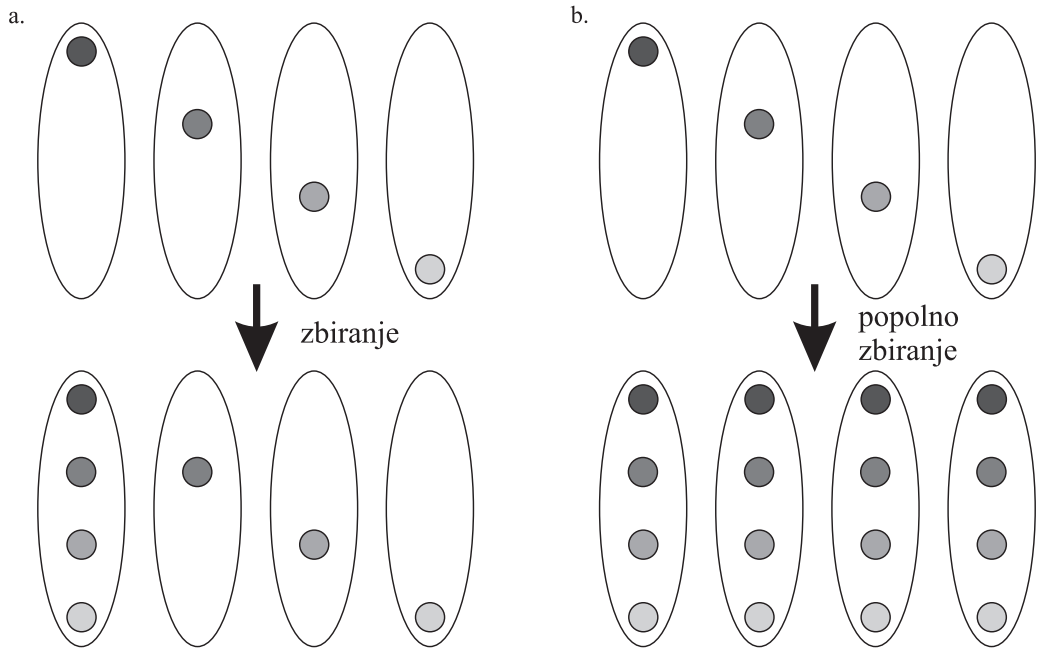
Vzemimo, da imamo v dvodimenzionalnem prostoru n delcev z enako maso. Vsak delec deluje na vse ostale delce z silo gravitacije, ki je odvisna od njegovega položaja in mase. V vsakem koraku simulacije obnašanja delcev v gravitacijskem polju je potrebno izračunati nov položaj in hitrost vsakega delca.

4.4.1 Delitev

Prvi korak snovanja algoritma je delitev niza podatkov o delcih. Vzemimo najprej, da imamo eno osnovno opravilo na delec. To opravilo mora določiti nov položaj in novo hitrost delca ob upoštevanju položaja in mase vseh ostalih delcev.

4.4.2 Komunikacija

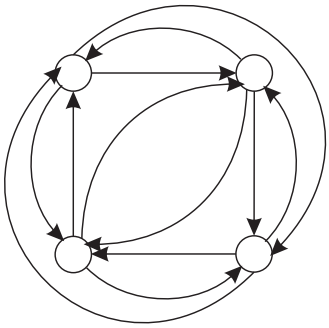
Operacija zbiranja (*ang.* gather) je globalna komunikacijska operacija, ki porazdeljeni podatkovni niz zbere na enem mestu (opravilu), tako kot prikazuje slika 4.11a. Za razliko od operacije redukcije, ki računa en sam rezultat na osnovi podatkov v



Slika 4.11: Operaciji a) zbiranja in b) zbiranja vseh.

nizu, operacija zbiranja pomeni povezovanje podatkov iz različnih opravil. Operacija popolnega zbiranja (*ang.* all-gather) je podobna operaciji zbiranja, le da ima ob koncu vsako opravilo kopijo celotnega podatkovnega niza, kot to prikazuje slika 4.11b.

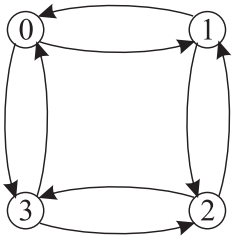
V obravnavanem problemu delcev v gravitacijskem polju za izračun novega položaja vsakega delca potrebujemo podatke o vseh ostalih delcih. Za to nalogo je zato zelo primerna operacija popolnega zbiranja. En način izvedbe je, da s kanali povežemo vse pare osnovnih opravil, tako kot na sliki 4.12. V vsakem koraku komunikacije z operacijo zbiranja vsako osnovno opravilo pošlje svoj niz podatkov drugemu opra-



Slika 4.12: Povezava vseh opravil z vsemi opravili.

vilu. Po $n-1$ komunikacijskih korakih ima vsako opravilo podatke o položaju vseh ostalih delcev in lahko izračuna svoj novi položaj in novo hitrost.

Ali obstaja hitrejši način, ki bi izvedel postopek zbiranja v logaritemskem številu komunikacijskih korakov? Če je v prvem koraku dolžina sporočila 1, v drugem 2, v tretjem 4, ..., ter v i-tem 2^{i-1}, potem je logaritemsko število korakov dovolj, da pridobimo vse podatke od ostalih opravil. Topologijo kanalov, ki ustreza tej shemi, lahko opišemo z grafom opravilo/kanal v obliki hiper-kocke (slika 4.13). Takšna topologija kanalov pogosto spremlja uspešno implementacijo izmenjav vsi-vsem (*ang.* all-to-all).



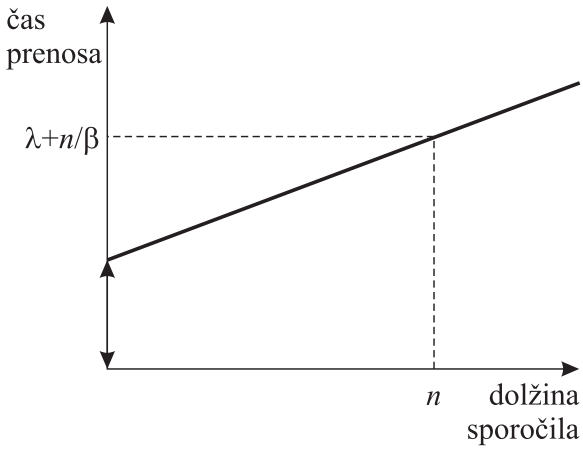
Slika 4.13: Hiperkocka s štirimi vozlišči.

4.4.3 Združevanje in preslikava

V splošnem je število delcev n vedno veliko večje od števila procesorjev p, $n \gg p$. Vzemimo, da je število n mnogokratnik števila p. Tedaj lahko preslikamo eno opravilo na procesor in v vsako opravilo združimo n/p delcev. Popolno zbiranje tedaj zahteva $\log_2 p$ komunikacijskih korakov. V prvem je dolžina sporočila n/p, v drugem $2n/p$, ...

4.4.4 Analiza

Zanima nas pričakovan čas izvajanja obravnavanega paralelnega algoritma. Dolžine sporočil so različne, zato moramo za natančno oceno vpeljati nove parametre: s parameterom λ bomo označevali latenco oziroma čas, potreben za pripravo prenosa, parameter β pa bo določal pasovno širino (*ang.* bandwidth) oziroma število podatkov v sporočilu, poslanih po kanalu v enoti časa. Čas, potreben za prenos sporočila z n podatki, je tako enak $\lambda + n/\beta$. Njegovo odvisnost od dolžine sporočila prikazuje tudi slika 4.14.



Slika 4.14: Čas prenosa v odvisnosti od dolžine sporočila.

Čas, potreben za prenos podatkov v eni iteraciji, ki predstavlja enkratno spremembo položajev in hitrosti vseh delcev, je enak

$$\sum_{i=1}^{\log_2 p} \left(\lambda + \frac{2^{i-1} n}{\beta p} \right) = \lambda \log_2 p + \frac{n(p-1)}{\beta p} \quad . \tag{4.3}$$

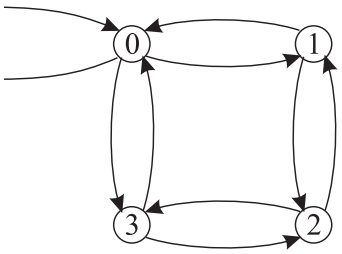
Vsako opravilo je zadolženo za izračun gravitacijske sile za n/p delcev. Vzemimo, da je čas računanje v vsaki iteraciji enak $\chi \frac{n}{p}$. Vsota časov, podanih v zgornjih dveh enačbah, je ravno pričakovan čas izvajanja ene iteracije v paralelnem programu

$$t_{\text{exe}} = \lambda \log_2 p + \frac{n(p-1)}{\beta p} + \chi \frac{n}{p} \quad . \tag{4.4}$$

4.4.5 Vhod in izhod

Do sedaj v modelu opravilo/kanal nismo upoštevali branja podatkov in izpisovanja rezultatov. V osnovni model lahko dodamo vhodno/izhodne kanale. Vzemimo,

da paralelni program na vhodu sprejema originalne položaje in hitrosti delcev. Večračunalniški sistemi običajno nimajo možnosti paralelnih vhodno/izhodnih operacij, zato bomo za vhodno/izhodne operacije namenili eno samo vhodno/izhodno opravilo, ki je povezano z enim samim običajnim opravilom (slika 4.15).



Slika 4.15: Vpeljava vhodno/izhodnih kanalov v model opravilo/kanal. Vhodni/izhodni kanali so dodani samo k opravilu z oznako 0. Računanje poteka na $p = 4$ procesorjih.

Ogledali si bomo, kako vhodno/izhodne operacije vplivajo na obravnavani primer obnašanja delcev v gravitacijskem polju. Vhodno/izhodno opravilo začne z odpiranjem datoteke in branjem položajev in hitrosti delcev. Predstavljeni problem je dvodimenzionalen, zato je vsak položaj delca in vsaka hitrost podana s po dvema komponentama. Če z izrazom $\lambda_{\text{io}}+n/\beta_{\text{io}}$ opišemo čas, potreben za vpis ali izpis n podatkov, potem zahteva branje vseh podatkov s po štirimi komponentami $\lambda_{\text{io}}+4n/\beta_{\text{io}}$ časa, pri čemer sta λ_{io} in β_{io} latenca in pasovna širina vhodno/izhodnega kanala.

Potem, ko vhodno/izhodno opravilo prebere podatke o vseh delcih, moramo podatke razdeliti na kose, od katerih vsak opisuje n/p delcev, in jih prenesti vsem opravilom. Takšni globalni komunikacijski operaciji rečemo raztros (*ang.* scatter). To je v bistvu operacija, ki je komplementarna zbiranju.

Vhodno/izhodno opravilo lahko raztrosi podatke po ostalih opravilih tako, da pošlje potrebne podatke vsakemu od ostalih opravil. Pošlje torej $p-1$ sporočil, vsako dolžine $4n/p$. Potreben čas za to je enak:

$$(p-1)\left(\lambda + 4\frac{n}{\beta p}\right) = (p-1)\lambda + 4\frac{n(p-1)}{\beta p} \quad . \tag{4.5}$$

To seveda ni dober algoritem, saj komunikacija, pri kateri eno opravilo po vrsti prenaša podate ostalim opravilom, procesorjev ne izkorišča uravnoteženo.

Če izvedemo pošiljanje tako kot prej v okviru binarnega drevesa ali hiper-kocke, potem to zahteva $\log_2 p$ komunikacijskih korakov. V prvem koraku vhodno/izhodno opravilo pošlje polovico podatkov drugemu opravilu. V drugem koraku obe opravili

s polovico podatki pošljeta četrtino podatkov preostalima opraviloma. Sedaj imajo vsa štiri opravila četrtino vseh podatkov. V tretjem koraku štiri opravila pošljejo osmino podatkov naslednjim štirim opravilom, in tako dalje. Celoten čas, potreben za komunikacijo, je v tem primeru enak

$$\sum_{i=1}^{\log_2 p} \left(\lambda + 4\frac{n}{2^i \beta p} \right) = \lambda \log_2 p + 4\frac{n(p-1)}{\beta p} \quad . \tag{4.6}$$

Očitno je drugi način komunikacije uspešnejši od prvega. Razlog za krajši čas, potreben za komunikacijo, v drugem primeru je v paralelnosti prenosov po različnih kanalih, ki nimajo skupnih izvornih ali ponornih opravil. To je možno doseči le v primeru stikalnih omrežij, ki dovoljujejo paralelno komunikacijo, ne pa tudi v primeru deljenih omrežij.

4.4.6 Celovita analiza

Sedaj je mogoče podati izraz za celotni čas, potreben za izračun ene iteracije obnašanja delcev v dvo-dimenzionalnem gravitacijskem polju. Pri tem sta vhodna in izhodna operacija, potrebni za prenos položajev in hitrosti delcev, izvedeni zaporedno

$$2(\lambda_{\text{io}} + 4\frac{n}{\beta_{\text{io}}}) \quad . \tag{4.7}$$

Čas, potreben za operacijo raztrosa podatkov o delcih na začetku in operacijo zbiranja podatkov o delcih na koncu, je enak

$$2(\lambda \log_2 p + 4\frac{n(p-1)}{\beta p}) \quad . \tag{4.8}$$

V vsaki iteraciji algoritma si opravila izmenjajo podatke o delcih, kar z uporabo operacije popolnega zbiranja vzame

$$\lambda \log_2 p + 2\frac{n(p-1)}{\beta p} \tag{4.9}$$

časa. V vsaki iteraciji vsak procesor izvede še svoj del računske naloge, za katero potrebuje čas, enak

$$\chi \frac{n}{p}(n-1) \quad . \tag{4.10}$$

V primeru, da izvedemo m iteracij, je pričakovani čas izvajanja algoritma enak

$$\begin{aligned} t_{\text{exe}} &= 2(\lambda_{\text{io}} + 4\frac{n}{\beta_{\text{io}}}) + 2(\lambda \log_2 p + 4\frac{n(p-1)}{\beta p}) + \\ & \quad m\left[\lambda \log_2 p + 2\frac{n(p-1)}{\beta p} + \chi\frac{n}{p}(n-1)\right] \quad . \end{aligned} \tag{4.11}$$

Poglavje 5

Performančna analiza

5.1 Uvod

Preden se lotimo kodiranja paralelnega programa, je dobro vedeti, kakšne bodo njegove lastnosti, saj je od tega lahko odvisno, ali se bomo sploh odločili za ta korak. S pomočjo analize časa izvajanja paralelnega programa lažje razumemo ovire za doseganje boljših rezultatov in spoznamo odvisnost časa izvajanja od števila procesorjev.

V nadaljevanju bomo najprej razvili splošno formulo za izračun pohitritve s paralelnim programom. Sledil bo pregled znanih formul za performančno analizo, kot so Amdahlov zakon, Gustafson-Barsisov zakon, Karp-Flattova metrika in metrika enake-uspešnosti. Amdahlov zakon nam pomaga pri odločitvi, ali je program sploh primeren za paralelizacijo. Gustafson-Barsisov zakon predstavlja način ocenjevanja paralelnega programa. S Karp-Flattovo metriko lahko ugotovimo, ali je ovira za večjo pohitritev obseg zaporedne kode ali strošek paralelizacije. Metrika enake uspešnosti omogoča ocenjevanje skalabilnosti paralelnega algoritma in tako pomaga pri iskanju rešitve, pri kateri se performanče lastnosti z večanjem števila procesorjev izboljšujejo.

5.2 Pohitritev in uspešnost

Razvoj in izvedbo paralelnih programov spremlja pričakovanje, da bo njihovo izvajanje hitrejše od ustrezne zaporedne različice. Zato je pohitritev definirana kot razmerje med časom izvajanja zaporednega algoritma in časom izvajanja paralelnega

algoritma

$$\text{pohitritev} = \frac{\text{čas izvajanja zaporednega algoritma}}{\text{čas izvajanja paralelnega algoritma}} \quad . \tag{5.1}$$

Operacije v paralelnem algoritmu lahko razvrstimo v tri skupine:

- operacije, ki se morajo izvajati zaporedno (v nadaljevanju zaporedne operacije),
- operacije, ki se lahko izvajajo paralelno (v nadaljevanju paralelne operacije) in
- operacije komunikacije in redundantnega računanja, ki predstavljajo strošek paralelizacije.

S temi skupinami lahko izdelamo preprost model pohitritve. Označimo z $\psi(n, p)$ pohitritev problema velikosti n s p procesorji, s $\sigma(n)$ čas potreben za izvajanje zaporednih operacij, s $\xi(n)$ čas, potreben za izvajanje paralelnih operacij, in s $\kappa(n, p)$ čas, ki predstavlja strošek paralelizacije.

Čisti zaporedni program za izvedbo na enem procesorju potrebuje $\sigma(n) + \xi(n)$ časa. Najboljši rezultat pri paralelni izvedbi istega programa je definiran kot vsota časov, potrebnih za izvedbo zaporednih operacij, $1/p$ paralelnih operacij in dodatka na račun paralelizacije, $\sigma(n) + \xi(n)/p + \kappa(n, p)$.

Pohitritev (*ang.* speedup) lahko sedaj zapišemo kot

$$\psi(n, p) \leq \frac{\sigma(n) + \xi(n)}{\sigma(n) + \xi(n)/p + \kappa(n, p)} \quad . \tag{5.2}$$

Z dodajanjem procesorjev zmanjšujemo čas izvajanja algoritma, vendar hkrati povečujemo čas, potreben za komunikacijo. Povečevanje števila procesorjev ni več smiselno, ko čas, potreben za komunikacijo, narašča bolj, kot se zmanjšuje čas računanja, saj se tedaj čas izvajanja paralelnega programa začne povečevati.

Uspešnost (*ang.* efficiency) paralelnega programa je mera za izkoriščenost procesorjev. Definirana je kot razmerje pohitritve in števila procesorjev

$$\begin{aligned} \text{uspešnost} &= \frac{\text{pohitritev}}{\text{število procesorjev}} \\ &= \frac{\text{čas izvajanja zaporednega algoritma}}{\text{čas izvajanja paralelnega algoritma} \times \text{število procesorjev}} \end{aligned} \tag{5.3}$$

Če uspešnost označimo kot $\varepsilon(n,p)$, potem jo lahko bolj formalno zapišemo z izrazom

$$\varepsilon(n,p) = \frac{\sigma(n)+\xi(n)}{(\sigma(n)+\xi(n)/p+\kappa(n,p))p} = \frac{\sigma(n)+\xi(n)}{\sigma(n)p+\xi(n)+\kappa(n,p)p} \quad . \tag{5.4}$$

Vsi členi v enačbi so večji ali enaki 0, zato velja $0 \leq \varepsilon(n,p) \leq 1$.

5.3 Amdahlov zakon

Izhajamo iz enačbe za pohitritev

$$\psi(n,p) \leq \frac{\sigma(n)+\xi(n)}{\sigma(n)+\xi(n)/p+\kappa(n,p)} \quad . \tag{5.5}$$

Ker je $\kappa(n,p) \geq 0$, lahko zapišemo

$$\psi(n,p) \leq \frac{\sigma(n)+\xi(n)}{\sigma(n)+\xi(n)/p+\kappa(n,p)} \leq \frac{\sigma(n)+\xi(n)}{\sigma(n)+\xi(n)/p} \quad . \tag{5.6}$$

Če s f označimo delež zaporednih operacij, ki jih ne moremo pohitriti,

$$f = \frac{\sigma(n)}{\sigma(n)+\xi(n)} \quad , \tag{5.7}$$

velja

$$\begin{aligned} \psi(n,p) &\leq \frac{\sigma(n)/f}{\sigma(n)+\sigma(n)\left(\frac{1}{f}-1\right)/)} \\ &\leq \frac{1/f}{1+\left(\frac{1}{f}-1\right)/p} \\ &\leq \frac{1}{f+\frac{1-f}{p}} \quad . \end{aligned} \tag{5.8}$$

Zadnji izraz je Amdahlov zakon, ki določa največjo možno pohitritev algoritma s paralelnim računalnikom s p procesorji. Zasnovan je na predpostavki, da želimo rešiti problem določene velikosti čim hitreje in postavlja zgornjo mejo pohitritve.

Na primer, pri $f = 10$ % je pohitritev z dvema procesorjema 1,82, z desetimi procesorji 5,26, s sto procesorji pa 9,17. Če gre število procesorjev p proti neskončnosti, gre pohitritev proti $1/f$ oziroma v našem primeru proti $1/0{,}1 = 10$. To tudi pomeni, da nujni sekvenčni del računanja omejuje pohitritev.

Kadar pri konstantnem številu procesorjev pohitritev narašča z velikostjo problema, govorimo o Amdahlovem efektu.

5.4 Gustafson-Barsisov zakon

Amdahlov zakon predpostavlja, da je minimizacija časa izvajanja algoritma osnovni cilj paralelnega računanja. Velikost problema obravnava kot konstanto in demonstrira, kako povečevanje števila procesorjev zmanjšuje čas izvajanja.

Pogosto pa je cilj paralelizacije povečanje natančnosti rešitve v določenem obsegu časa. Če torej obravnavamo čas izvajanja algoritma kot konstanto in predpostavljamo, da se velikost problema povečuje s številom procesorjev, potem se delež zaporednih operacij z naraščanjem velikosti problema zmanjšuje (Amdahlov efekt). Povečevanje števila procesorjev omogoča povečevanje problema, zmanjševanje deleža zaporednih operacij in večjo pohitritev.

Pohitritev ob predpostavki $\kappa(n,p) \geq 0$ je enaka

$$\psi(n,p) \leq \frac{\sigma(n)+\xi(n)}{\sigma(n)+\xi(n)/p} \quad . \tag{5.9}$$

Označimo s s delež časa, ki gre v paralelnem algoritmu na račun zaporednih operacij, in z $1-s$ delež časa, v katerem se operacije izvajajo paralelno,

$$s = \frac{\sigma(n)}{\sigma(n)+\xi(n)/p} \tag{5.10}$$

$$1-s = \frac{\xi(n)/p}{\sigma(n)+\xi(n)/p} \quad . \tag{5.11}$$

S preureditvijo zgornjih izrazov dobimo

$$\begin{aligned} \sigma(n) &= (\sigma(n)+\xi(n)/p)s \quad , \\ \xi(n) &= (\sigma(n)+\xi(n)/p)(1-s)p \quad . \end{aligned} \tag{5.12}$$

Če zadnja izraza vstavimo v izraz za pohitritev, dobimo:

$$\begin{aligned} \psi(n,p) &\leq \frac{\sigma(n)+\xi(n)}{\sigma(n)+\xi(n)/p} = \frac{(\sigma(n)+\xi(n)/p)(s+(1-s)p)}{\sigma(n)+\xi(n)/p} \\ &\leq s+(1-s)p = p+(1-p)s \quad . \end{aligned} \tag{5.13}$$

Gustafson-Barsisov zakon določa največjo možno pohitritev danega paralelnega programa, s katerim rešujemo problem velikosti n z uporabo p procesorjev, pri čemer s označuje delež zaporednih operacij v celotnem času izvajanja pralelnega algoritma.

Medtem ko Amdahlov zakon določa pohitritev tako, da vzame za izhodišče zaporedno računanje in napove hitrost računanja na večprocesorskem sistemu, Gustafson-Barsisov zakon deluje ravno obratno. Začne s paralelnim računanjem in oceni koliko hitrejše je lahko paralelno računanje od zaporednega za isti problem. Amdahlov zakon je pesimističen in velja dobro za programe, kjer ne moremo izboljšati zaporednega deleža, Gustafson-Barsisov zakon pa je optimističen in velja dobro za programe, kjer se obseg podatkov manjša s številom procesorjev. Na primer, za $s = 10$ % bi v primeru dveh procesorjev dobili pohitritev 1,90, pri desetih procesorjih 9,1, pri 100 processorjih pa 90. V primeru, ko število procesorjev narašča proti neskončno, bi bila pohitritev približno enako številu procesorjev, $\psi(n,\infty) \approx p$.

5.5 Karp-Flattova metrika

Ocena pohitritve po Amdahlovem in Gustafson-Barsisovem zakon je precenjena, saj zanemarjata strošek paralelizacije $\kappa(n,p)$. Zato sta Karp in Flatt predlagala drugačno metriko, ki se imenuje eksperimentalno določen zaporedni delež (*ang.* experimentally determined serial fraction), in omogoča boljši vpogled v performančno analizo.

Do sedaj smo opisovali čas izvajanja paralelnega programa na paralelni arhitekturi s p procesorji z izrazom:

$$T(n,p) = \sigma(n) + \xi(n)/p + \kappa(n,p) \quad , \tag{5.14}$$

kjer je $\sigma(n)$ čas, potreben za izvedbo zaporednih operacij, $\xi(n)$ čas, potreben za izvedbo paralelnih operacij, in $\kappa(n,p)$ presežek paralelizacije, ki izhaja iz komunikacije, sinhronizacije in redundantnega računanja. Ker zaporedni algoritem nima komunikacijskega presežka, je njegov čas izvajanja enak

$$T(n,1) = \sigma(n) + \xi(n) \quad . \tag{5.15}$$

Eksperimentalno določen zaporedni delež e paralelnega algoritma določa delež časa, ki se v paralelnem algoritmu porabi za izračun nezaželenih operacij (zaporedne operacije in operacije zaradi paralelizacije), merjenega glede na čas izvajanja zaporednega algoritma,

$$e = \frac{\sigma(n) + \kappa(n,p)}{T(n,1)} \quad . \tag{5.16}$$

Algoritem je tem boljši, čim manjši je e.

Ob upoštevanju relacij $\sigma(n) + \kappa(n,p) = T(n,1)e$ in $\xi = T(n,1) - \sigma(n) = T(n,1) - T(n,1)e$, kjer smo upoštevali, da je $\kappa(n,1) = 0$, lahko zapišemo čas izvajanja paralelnega algoritma kot

$$T(n,p) = T(n,1)e + T(n,1)(1-e)/p \quad . \tag{5.17}$$

Ker velja $\psi(n,p) = T(n,1)/T(n,p)$ oziroma $T(n,1) = T(n,p)\psi(n,p)$, veljajo ob predpostavki $\kappa(n,p)/p \approx 0$ naslednje zveze

$$\begin{aligned} T(n,p) &= T(n,p)\psi(n,p)e + T(n,p)\psi(n,p)(1-e)/p \\ 1 &= \psi(n,p)e + \psi(n,p)(1-e)/p \\ 1/\psi(n,p) &= e + (1-e)/p \\ 1/\psi(n,p) &= e + 1/p - e/p \\ e &= \frac{1/\psi(n,p) - 1/p}{1 - 1/p} \quad . \end{aligned} \tag{5.18}$$

Iz enačbe 5.18 vidimo, da eksperimentalno določeni zaporedni delež e lahko enostavno določimo z meritvijo pohitritev pri različnem številu procesorjev p.

Karp-Flattova metrika je pomembna iz dveh razlogov. Prvič, upošteva presežek zaradi paralelizacije $\kappa(n,p)$, ki ga Amdahlov in Gustafson-Barsisov zakon ignorirata, in drugič, s pomočjo te metrike je mogoče določiti razloge, ki onemogočajo boljšo paralelizacijo algoritma. Karp-Flattova metrika predpostavlja, da p procesorjev izvede paralelni del računanja p krat hitreje kot en procesor, zaradi česar preide člen $\xi(n)$ iz $T(n,1)$ v člen $\xi(n)/p$ v $T(n,p)$. S tem se ne upošteva možnosti, da se delo lahko porazdeli med procesorje tudi neenakomerno, kar poveča čas računanja.

Pri problemu konstantne velikosti se učinkovitost paralelnega računanja zmanjšuje s povečevanjem števila procesorjev. Karp-Flattova metrika vključuje tako čas, potreben za izračun zaporednih operacij kot tudi časovni presežek zaradi paralelizacije. Z uporabo eksperimentalno določenega zaporednega deleža e zato lahko določimo, ali

se uspešnost zmanjšuje zaradi omejenih možnosti paralelizma (zaporedne operacije) ali zaradi presežkov zaradi paralelizacije.

V tabeli 5.5 so podane pohitritve $\psi(n,p)$ in eksperimentalno določeni zaporedni delež e v odvisnosti od števila procesorjev p za dva paralelna algoritma.

Tabela 5.1: Pohitritve ψ in eksperimentalno določeni zaporedni delež e v odvisnosti od števila procesorjev p za dva paralelna algoritma.

algoritem	p	2	3	4	5	6	7	8
1	ψ	1,82	2,50	3,08	3,57	4,00	4,38	4,71
	e	0,1	0,1	0,1	0, 1	0,1	0,1	0,1
2	ψ	1,87	2,61	3,23	3,73	4,14	4,46	4,71
	e	0,07	0,07	0,08	0,08	0,09	0,09	0,1

V prvem primeru e ne narašča s p, zato je razlog za skromno pohitritev omejena možnost paralelizma (velik del računskih operacij je nujno zaporednih). V drugem primeru pa e postopno narašča, kar pomeni, da je razlog za skromno pohitritev presežek zaradi paralelizacije. Razlogi so lahko v času, potrebnem za zagon procesa, komunikacija, sinhronizacija ali pa arhitekturne omejitve.

5.6 Metrika enake uspešnosti

Videli smo že, da je skalabilnost paralelnega sistema (paralelnega programa na paralelnem računalniku) mera njegove sposobnosti, da izboljša obnašanje s povečanjem števila procesorjev.

Očitno sta pohitritev in uspešnost naraščajoči funkciji velikosti problema, saj je komunikacijska kompleksnost običajno manjša od računske kompleksnosti. Temu smo rekli Amdahlov efekt. Da ohranimo enak nivo uspešnosti, medtem ko dodajamo nove procesorje, pa lahko povečamo tudi velikost problema. Ta ideja je formalizirana v metriki enake uspešnosti.

Originalno definicijo za pohitritev

$$\psi(n,p) \leq \frac{\sigma(n)+\xi(n)}{\sigma(n)+\xi(n)/p+\kappa(n,p)} \tag{5.19}$$

zapišimo malo drugače

$$\psi(n,p) \leq \frac{p(\sigma(n)+\xi(n))}{p\sigma(n)+\xi(n)+p\kappa(n,p)}$$

$$\leq \frac{p(\sigma(n)+\xi(n))}{\sigma(n)+\xi(n)+(p-1)\sigma(n)+p\kappa(n,p)} \quad . \tag{5.20}$$

Označimo s $\tau(n,p) = (p-1)\sigma(n) + p\kappa(n,p)$ čas, potreben za izvedbo operacij, ki jih v zaporednem algoritmu ni. Enačba 5.20 se s tem poenostavi v

$$\psi(n,p) \leq \frac{p(\sigma(n)+\xi(n))}{\sigma(n)+\xi(n)+\tau(n,p)} \quad . \tag{5.21}$$

Ker je uspešnost enaka pohitritvi deljeno s p, dobimo

$$\begin{aligned} \varepsilon(n,p) = \frac{\psi(n,p)}{p} &\leq \frac{\sigma(n)+\xi(n)}{\sigma(n)+\xi(n)+\tau(n,p)} \\ &\leq \frac{1}{1+\frac{\tau(n,p)}{\sigma(n)+\xi(n)}} \\ &\leq \frac{1}{1+\frac{\tau(n,p)}{T(n,1)}} \quad . \end{aligned} \tag{5.22}$$

Zgornjo neenačbo lahko nadalje preoblikujemo v

$$\frac{\tau(n,p)}{T(n,1)} \leq \frac{1-\varepsilon(n,p)}{\varepsilon(n,p)} \tag{5.23}$$

oziroma v

$$T(n,1) \geq \frac{\varepsilon(n,p)}{1-\varepsilon(n,p)}\tau(n,p) \quad . \tag{5.24}$$

Če želimo ohraniti konstantni nivo uspešnosti, potem mora biti člen $\frac{\varepsilon(n,p)}{1-\varepsilon(n,p)}$ konstanten in zadnji izraz se poenostavi v

$$T(n,1) \geq C\tau(n,p) \quad , \quad C \in \mathbf{R} \quad . \tag{5.25}$$

Za ohranitev enakega nivoja uspešnosti pri povečevanju števila procesorjev se mora velikost problema n povečevati tako, da je izpolnjena zgornja neenačba.

Poglavje 6

Paralelno programiranje s knjižnico MPI

Standard MPI (*ang.* Message Passing Interface) je trenutno najpopularnejše programsko orodje za paralelno programiranje, [16, 19, 20]. Kot pove že samo ime knjižnice, poteka komunikacija med procesi z izmenjevanjem sporočil. Skoraj vsak računalnik, dostopen na trgu, podpira standard MPI, poleg tega brezplačne knjižnice MPI sledijo standardom MPI, kar omogoča delo na poljubnem paralelnem računalniškem sistemu, tudi takšnem, ki je sestavljen iz univerzalnih delov.

Pri programiranju s pomočjo knjižnice MPI se bomo omejili na programski jezik C. V tem poglavju si bomo najprej ogledali elementarne funkcije MPI, nato bo podan način prevajanja in izvajanja paralelnih programov, ki vključujejo funkcije MPI, sledil bo opis funkcij za kolektivno komuniciranje in zatem še opis funkcij, ki so v pomoč pri performančni analizi programov.

6.1 Elementarne funkcije MPI

Za vse funkcije MPI velja, da se začnejo s predpono `MPI_`, ki ji sledi ime funkcije v katerem je prva črka velika. Podobno se vse konstante iz knižnjice MPI začnejo s predpono `MPI_`, ki ji sledi niz velikih črk.

V razred elementarnih funkcij sodijo funkcije:

- `MPI_Init`: inicializira proces MPI, zato mora biti prva med funkcijami MPI v programu,
- `MPI_Finalize`: konča proces MPI, zato mora biti zadnja izmed funkcij MPI

v programu,

- `MPI_Comm_size`: določa število procesov (opravil),
- `MPI_Comm_rank`: določa oznako procesa (procesorja),
- `MPI_Send`: pošlje sporočilo točno določenemu procesu,
- `MPI_Recv`: sprejme sporočilo od točno določenega procesa.

6.1.1 `MPI_Init`

Funkcija inicializira proces MPI. Čeprav mora biti v programu prva izmed funkcij iz knjižnice MPI, pa ni potrebno, da je absolutno prva v programu.

Sintaksa: `MPI_Init(&argc, &argv);`

6.1.2 `MPI_Finalize`

To je zadnja izmed funkcij MPI v programu in zaključi proces MPI in s tem sprosti vire, na primer pomnilnik, ki so bili dodeljeni procesu MPI.

Sintaksa: `MPI_Finalize();`

6.1.3 `MPI_Comm_size`

Po inicializaciji postane vsak aktivni proces MPI član objekta imenovanega komunikator (*ang.* communicator), ki vzpostavlja okolje za prenašanje sporočil med procesi. Komunikator `MPI_COMM_WORLD` je standardni komunikator, ki vključuje vse aktivne procese. Mogoče je vzpostaviti tudi lastne komunikatorje, predvsem, če želimo aktivne procese razdeliti v več neodvisnih komunikacijskih skupin. Procesi v komunikatorju so urejeni tako, da rang procesa (*ang.* rank) določa njegovo oznako ali identiteto (*ang.* ID). V komunikatorju, ki združuje p procesov, obstajajo rangi od 0 do $p - 1$. Z uporabo ranga lahko z istim programom različnim procesom dodelimo različne naloge in/ali različne dele podatkov (*ang.* Single Program Multiple Data, SPMD).

Proces s klicem te funkcije izve število aktivnih procesov v komunikatorju.

Sintaksa: `MPI_Comm_size (MPI_COMM_WORLD, &p);`

6.1.4 MPI_Comm_rank

S klicem funkcije `MPI_Comm_rank` proces pridobi svoj rang znotraj komunikatorja.

Sintaksa: `MPI_Comm_rank(MPI_COMM_WORLD, &id);`

6.1.5 MPI_Send

To je ena izmed dveh funkcij, ki omogočata prenašanje sporočil med dvema procesoma oziroma točkama (*ang.* point-to-point communication). Pri točkovni komunikaciji proces i pošlje sporočilo procesu j, medtem ko ostali procesi ne sodelujejo. Na sliki 6.1 je primer programske kode, ki s pogojnimi stavki vzpostavi točkovno komunikacijo.

```
if (id == i)
    //pošlji sporočilo procesu j
else if (id==j)
    //sprejmi sporočilo od procesa i
```

Slika 6.1: Točkovna komunikacija med procesoma i in j.

Sintaksa:
```
MPI_Send(void *message,
         int count,
         MPI_Datatype datatype,
         int destination,
         int tag,
         MPI_Comm comm);
```

Parameter `message` je kazalec na polje podatkov (sporočila), ki jih izvorni proces s pomočjo te funkcije prenaša drugemu procesu. Parameter `count` določa število podatkovnih enot, parameter `datatype` pa je tip podatkovne enote. Funkcija MPI zaradi večje združljivosti programov z različnimni operacijskimi sistemi definira svoje podatkovne tipe, na primer `MPI_CHAR`, `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`. Vsi podatki morajo biti istega tipa. Parameter `destination` je rang ponornega procesa, ki mu je sporočilo namenjeno. Parameter `tag` je celoštevilčna oznaka sporočila, ki omogoča različno namembnost sporočila. Parameter `comm` označuje komunikator, v katerem je sporočilo poslano.

Funkcija `MPI_Send` se ne zaključi, dokler sporočilni vmesnik (*ang.* buffer) ni ponovno dostopen. Zato se običajno sporočilo kopira v sistemski sporočilni vmesnik, kar omogoča, da funkcija `MPI_Send` hitro vrne kontrolo procesu, ki jo je klical.

6.1.6 MPI_Recv

To je druga funkcija za točkovno komunikacijo in omogoča sprejem (*ang.* receive) sporočila.

Sintaksa:
```
MPI_Recv(void *message,
         int count,
         MPI_Datatype datatype,
         int source,
         int tag,
         MPI_Comm comm,
         MPI_Status *status);
```

Parameter `message` je kazalec na podatkovno strukturo, v katero naj se sporočilo shrani. Parameter `count` označuje maksimalno število podatkovnih enot, ki jih je proces pripravljen sprejeti, parameter `datatype` določa tip podatkovnih enot, parameter `source` je rang izvornega procesa, ki pošilja sporočilo, parameter `tag` je oznaka sporočila, ki ga proces želi sprejeti, parameter `comm` pa določa komunikator, v katerem se prenaša sporočilo. Parameter `status` je kazalec na zapis tipa tipa `MPI_Status`, v katerem funkcija vrne status, na primer vzrok napake. `MPI_Status` je edina podatkovna struktura v knjižnici MPI, ki je dostopna uporabniku.

Funkcija `MPI_Recv` se izvaja, dokler sporočilo ni sprejeto, ali dokler ne pride do napake. Ko funkcija vrne kontrolo klicočemu procesu, zapis `status` vsebuje naslednje podatke:

- `status->MPI_source`: rang procesa, ki pošilja sporočilo,
- `status->MPI_tag`: oznako sporočila
- `status->MPI_ERROR`: vzrok napake

Parameter `status` vrne tudi informacijo o velikosti sporočila, vendar ta ni direktno dosegljiva uporabniku. Velikost sporočila je mogoče dobiti s klicem funkcije

```
int MPI_Get_count(MPI_Status *status,
                  MPI_Datatype datatype,
                  int *count_ptr);
```

Funkciji `MPI_Send` in `MPI_Recv` vračata kode napak tipa `int`. Običajno napaka funkcije MPI prekine izvrševanje programa.

6.1.7 Sporočila

Procesi med seboj komunicirajo tako, da si izmenjujejo sporočila. Sporočilo sestavljajo ovojnica in podatki. Ovojnica mora vsebovati najmanj rang izvornega in rang ponornega procesa, velikost in oznako sporočila ter komunikator.

6.2 Prevajanje in izvajanje programov MPI

Čeprav smo si do sedaj ogledali le šest najosnovnejših funkcij MPI, podajmo pred nadaljevanjem opisa knjižnice MPI način pisanja, prevajanja in izvajanja MPI programov.

Enostaven program, v katerem procesi z rangi od 1 do $p-1$ pošljejo niz znakov `Hello world!` procesu z rangom 0, je predstavljen na sliki 6.2.

Program vključuje direktivo `#include <mpi.h>`, ki vanj vključi funkcije in konstante iz knjižnice MPI. Funkcija `main` mora vsebovati argumenta `argv` in `argc` saj preko njiju iz ukazne vrstice nastavljamo, kako naj se inicializirajo procesi MPI. Pred funkcijo `MPI_Init` in za funkcijo `MPI_Finalize` ne smemo poklicati nobene funkcije MPI. V programu je predstavljen tudi koncept enega programa za več procesorjev (SPMD). Glede na rang procesa se v enem primeru izvaja sprejemanje in izpisovanje sporočil, v vseh ostalih pa pošiljanje.

Potem, ko shranimo program v datoteko, na primer `hello.c`, sledi prevajanje izvorne kode v izvršilno obliko. Ukaz za prevajanje se lahko razlikuje od sistema do sistema in od razvojnega okolja do razvojnega okolja.

Za zaganjanje izvršilnih datotek, ki uporabljajo funkcije MPI, je knjižnici MPI priložen program `mpirun` ali `mpiexec`. Z enostavnim klicem `mpiexec -np <število procesov> <program>` zaženemo program `<program>` s `<število procesov>` procesi. Z dodatnimi argumenti lahko prilagodimo okolje svojim željam, na primer, določimo lahko, na katerih računalnikih naj se program izvaja. Za mnoge operacijske sisteme so na voljo tudi grafični vmesniki, ki olajšajo nastavljanje parametrov programa `mpiexec`.

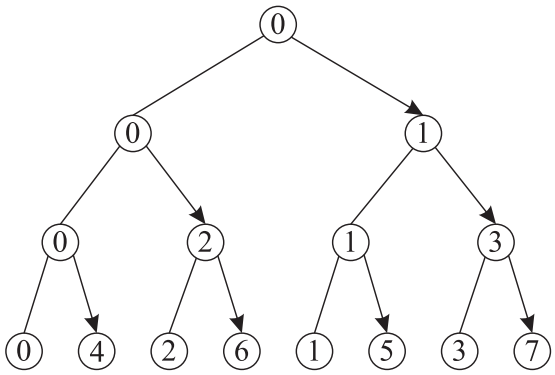
6.3 Funkcije za kolektivno komuniciranje

V primeru, ko en proces pošilja podatke ostalim procesom s točkovno komunikacijo, pride do precejšnje neaktivnosti procesov in posledično do slabe učinkovitosti računalniškega sistema. Komunikacijo in uspešnost izboljšamo s kolektivno komuni-

```
#include <mpi.h>
#include <string.h>

main(int argc, char *argv[])"
{
    int myRank;         /* rang procesa */
    int numProcs;       /* število procesov */
    int source;         /* rang pošiljatelja */
    int destination = 0;/* rang prejemnika */
    int tag = 0;        /* oznaka sporočila */
    char message[100];  /* rezervacija prostora za sporočilo */
    MPI_Status status;  /* status funkcije MPI_Receive */

    MPI_Init(&argv, &argc);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    if( myRank != 0 )
    {
        strcpy(message, "Hello world!");
        MPI_Send(message, strlen(message)+1, MPI_CHAR,
                 destination, tag, MPI_COMM_WORLD);
    }
    else
    {
        for( source = 1; source < numProcs; source++)
        {
            MPI_Recv(message, 100, MPI_CHAR,
                     source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize();
}
```

Slika 6.2: Program, ki uporablja elementarne funkcije MPI.

kacijo. Tedaj komunikacijski vzorec zajema vse procese v komunikatorju. Slika 6.3 prikazuje, kako lahko s komunikacijo v topologiji binarnega drevesa precej pohitrimo prenos podatkov od enega procesa do ostalih. Na sliki je predstavljeno delovanje funkcije za kolektivno komunikacijo. Oznake v vozliščih pomenijo rang procesov, vsak nivo drevesa pa podaja po vrsti korake komuniciranja in določa procese, ki so v določenem koraku aktivni. Na korenskem nivoju (nivo 0) pošlje proces 0 sporočilo



Slika 6.3: Binarno drevo za kolektivno komunikacijo.

procesu 1. Na prvem nivoju pošlje proces 0 podatke procesu 2, proces 1 pa procesu 3, na drugem nivoju pošlje proces 2 sporočilo procesu 6, proces 1 procesu 5 in proces 3 procesu 7. Na primer, če bi imeli $p = 1024$ procesov, bi za prenos sporočila od enega procesa k vsem ostalim s točkovno komunikacijo potrebovali $p - 1 = 1023$ korakov, v primeru komunikacije po binarnem drevesu pa $\lceil \log_2 p \rceil = 10$, torej za dva velikostna reda manj.

V nadaljevanju bodo opisane najbolj pogoste funkcije MPI za kolektivno komunikacijo, `MPI_Bcast` (oddaja), `MPI_Reduce` (operacija redukcije), `MPI_Gather` (zbiranje), `MPI_ Scatter` (raztros) ter `MPI_Allgather` (popolno zbiranje) in `MPI_Allreduce` (popolna operacija redukcije).

6.3.1 `MPI_Bcast`

Oddaja (*ang.* broadcast) je kolektivna komunikacija, pri kateri posamezni proces pošilja isto sporočilo (podatke) vsem ostalim procesom v komunikatorju. V zgornjem primeru binarnega drevesa je bil čas za prenos podatkov od enega procesa do ostalih bistveno krajši kot v primeru točkovne komunikacije. Vendar pa je takšna komunikacija programsko bolj zahtevna, saj je potrebno v vsakem koraku opredeliti aktivne procese in določiti, kateri oddajajo in kateri sprejemajo sporočilo. Na primer, za oznake procesov s slike 6.3 (možne so tudi druge oznake), bi veljalo:

- nivo 0: $0 \rightarrow 1$,
- nivo 1: $0 \rightarrow 2$, $1 \rightarrow 3$,
- nivo 2: $0 \rightarrow 4$, $1 \rightarrow 5$, $2 \rightarrow 6$, $3 \rightarrow 7$.

kar lahko prevedemo v logične pogoje

```
if( 2^nivo ≤ rang < 2^nivo+1 )
    sprejem sporočila od procesa:  rang - 2^nivo
if( rang < 2^nivo )
    oddajanje sporočila procesu:  rang + 2^nivo
```

S pomočjo zgornjih logičnih pogojev o aktivnih procesih v posameznih korakih komunikacije, je programska izvedba prenosov lahko podana z zanko:

```
for( nivo = 0; nivo ≤ last; nivo++)
    if sprejemam(nivo, rang, &vir)
        MPI_Receive(podatki, ..., vir, ...)
else if( oddajam(nivo, rang, &ponor) )
        MPI_Send(podatki, ..., ponor, ...)
```

Funkcija `sprejemam` vrne 1, če je na trenutnem nivoju klicani proces sprejemal podatke, sicer vrne 0. V primeru, da vrne 1, se v parameter vir vpiše rang pošiljatelja. Funkcija `oddajam` podobno vrne 1, če proces `rang` na trenutnem nivoju pošilja podatke, sicer 0. V primeru, da funkcija vrne 1, je v parametru `ponor` zapisan rang prejemnika sporočila. Obe funkciji izhajata neposredno iz pogojev, ki so v psevdo kodi podani zgoraj.

Pri programiranju komunikacijskih funkcij je potrebno dobro poznati topologijo sistema, na katerem se bo izvajal paralelni program. Brez tega ni mogoče vedeti, ali je uporabljana programska koda komunikacije najboljša. To pa tudi pomeni, da je mogoče funkcije MPI, ki imajo specificirane le sintakse funkcijskih klicov in rezultatov, prilagoditi dejanski topologiji večračunalniškega sistema in tako izboljšati lastnosti paralelnih programov.

Sintaksa:
```
MPI_Bcast(void *message,
          int count,
          MPI_Datatype datatype,
          int root,
          MPI_Comm comm);
```

Ta funkcija pošilja sporočilo `message` od korenskega procesa z rangom `root` do vseh ostalih procesov v komunikatorju `comm`. Pri tem se za sprejem ne uporablja funkcija `MPI_Recv`, ampak se kliče funkcija `MPI_Bcast` z vseh procesov v komunikatorju z enakimi argumenti za paramtera `root` in `comm`. Parametra `count` in `datatype` imata enak pomen kot pri funkcijah `MPI_Send` in `MPI_Recv`. Ker parameter `tag` pri funkcijah za kolektivno komunikacijo ni prisoten, sta parametra `count` in `datatype`

enaka za vse procese. Pri funkciji `MPI_Bcast` parameter `message` pri korenskem procesu predstavlja izhod, pri ostalih procesih pa vhod.

6.3.2 `MPI_Reduce`

Operacija redukcije (*ang.* reduce) je kolektivna komunikacija, kjer vsi procesi v komunikatorju prispevajo svoje podatke, shranjene v pomnilniku procesa na naslovu, na katerega kaže kazalec `operand`. Podatki se nato združijo z asociativno binarno operacijo `operator` in se shranijo na pomnilniški naslov, na katerega kaže kazalec `result` v procesorju z rangom `root`. Asociativne binarne operacije so na primer seštevanje, množenje, največja vrednost, najmanjša vrednost, logični ali, logični in.

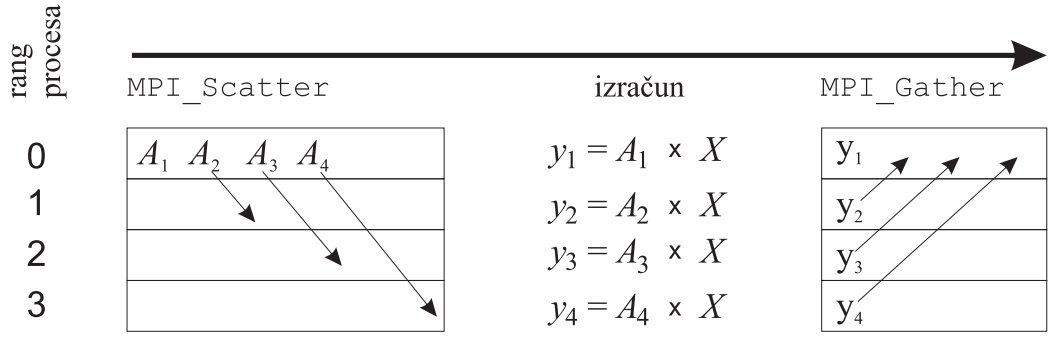
Bolj smiselno, kot da en proces sprejema delne rezultate ali podatke in nato nad njimi naredi operacijo redukcije, je, da za komunikacijo uporabimo obratno shemo kot pri razpošiljanju podatkov, torej, v primeru binarnega drevesa od listov proti korenu. Pri tem vsak proces kliče funkcijo `MPI_Reduce` z enakimi argumenti.

Sintaksa:
```
MPI_Reduce(void *operand,
           void *result,
           int count,
           MPI_Datatype datatype,
           MPI_Op operator,
           int root,
           MPI_Comm comm);
```

6.3.3 `MPI_Scatter` in `MPI_Gather`

Za prenašanje in sprejemanje delov programskih struktur med procesi sta namenjeni funkcija raztrosa `MPI_Scatter` in funkcija zbiranja `MPI_Gather`.

Vzemimo, da želimo paralelizirati množenje matrike z vektorjem, $A_{m\times n} \cdot X_{n\times 1} = Y_{m\times 1}$. Pri tem so elementi vektorja Y pravzaprav skalarni produkti vrstic matrike A in vektorja X. Vzemimo, da imajo sliko vektorja X že vsi procesi, ki bodo računali skalarne produkte. S funkcijo `MPI_Scatter` enostavno razstrosimo vrstice matrike A vsem omenjenim procesom, po končanem računanju skalarnih produktov pa s funkcijo `MPI_Gather` sestavimo vektor Y na korenskem procesu (slika 6.4). Vzemimo, da matriko A namesto po vrsticah raztrosimo po stolpcih, namesto, da do procesov prenašamo celoten vektor X, pa med procesi ustrezno razstrosimo njegove vrstice. Vsak proces v tem primeru lahko izračuna delni skalarni produkt. Vektor Y dobimo tako, da vse delne produkte seštejemo, za kar bi lahko uporabili funkcijo `MPI_Reduce`.



Slika 6.4: Shematičen prikaz uporabe funkcij `MPI_Scatter` in `MPI_Gather` pri računanju produkta matrike $A_{4\times n}$ in vektorja $X_{n\times 1}$ na $p = 4$ procesih. Z A_i je označena i-ta vrstica matrike A.

Sintaksa:
```
MPI_Scatter(void *send_data,
            int send_count,
            MPI_Datatype send_type,
            void *recv_data,
            int recv_count,
            MPI_Datatype recv_type,
            int root,
            MPI_Comm comm);
```

Funkcija `MPI_Scatter` razdeli podatke na naslovu `send_data` v pomnilniku korenskega procesorja (z rangom `root`) na p segmentov, od katerih vsak vsebuje `send_count` elementov tipa `send_type`. Prvi segment je poslan procesu z rangom 0, drugi procesu z rangom 1, in zadnji procesu z rangom $p - 1$. Parametra `root` in `comm` morata biti enaka na vseh procesih. Namesto kazalcev, ki jih ne poznamo, na primer `send_data` na procesu, ki ni korenski, lahko uporabimo konstanto `NULL`.

Sintaksa:
```
MPI_Gather(void *send_data,
           int send_count,
           MPI_Datatype send_type,
           void *recv_data,
           int recv_count,
           MPI_Datatype recv_type,
           int root,
           MPI_Comm comm);
```

Ta funkcija zbira podatke, ki so shranjeni v vseh procesorjih na naslovih `send_data`, ter jih shrani v urejenem redu na naslovu `recv_data` procesorja z rangom `root`.

6.3.4 `MPI_Allgather` in `MPI_Allreduce`

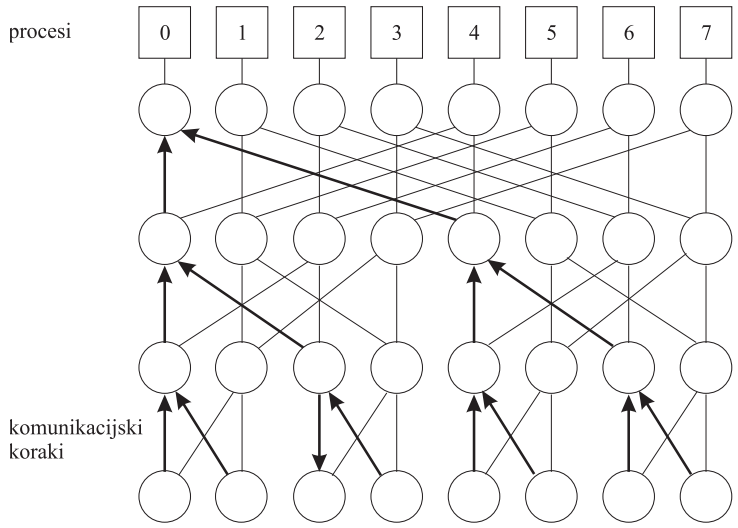
Velikokrat rezultat izračuna, na primer, produkt matrike in vektorja iz prejšnjega primera, za nadaljnje delo potrebujejo vsi procesi. V takih primerih je smiselno uporabiti funkcijo popolnega zbiranja `MPI_Allgather`, ki podatke, ki jih procesi shranjujejo na naslovih `send_data`, sestavi in jih shrani v vse procese na naslov `recv_data`. Funkcija torej simultano združi vse elemente vektorja X na vseh procesih, ne le na korenskem. Klic funkcije je podoben klicu funkcije `MPI_Gather`, le da ne vsebuje parametra `root`, saj rezultate sprejmejo vsi procesi.

Sintaksa:
```
MPI_AllGather(void *send_data,
              int send_count,
              MPI_Datatype send_type,
              void *recv_data,
              int recv_count,
              MPI_Datatype recv_type,
              MPI_Comm comm);
```

Tako kot funkcija `MPI_Allgather` predstavlja razširitev funkcije `MPI_Gather`, predstavlja funkcija `MPI_Allreduce` posplošitev funkcije `MPI_Reduce`.

Sintaksa:
```
MPI_AllReduce(void *operand,
              void *result,
              int count,
              MPI_Datatype datatype,
              MPI_Op operator,
              MPI_Comm comm);
```

Za učinkovito izvedbo funkcije `MPI_Allreduce` potrebujemo za vsak proces komunikacijo, ki sledi topologiji binarnega drevesa. Izkaže se, da hkratna komunikacija



Slika 6.5: Binarno drevo procesa 0 v topologiji metulja.

med vsemi procesi sledi topologiji metulja, ki smo jo že spoznali v poglavju 3.4.1. Na sliki 6.5 so poudarjeni koraki, ki prikazujejo pot podatkov do procesa z rangom 0.

Na sliki 6.6 je prikazano jedro programa za množenje matrike in vektorja, v katerem so uporabljene funkcije za kolektivno komunikacijo. Program je zasnovan tako, da je vektor Y po množenju na voljo na vseh procesih za nadaljnjo uporabo.

Da nam ob spremembi velikosti struktur ni potrebno ponovno prevajati programa, je pripravno pomnilniški prostor rezervirati dinamično med izvajanjem programa. V programskem jeziku C definiramo kazalec na strukturo, pomnilniški prostor pa rezerviramo s funkcijo `malloc`. Rezervacija enodimenzionalnega vektorja dolžine n izgleda takole

```
float* X;

X = (float*)malloc(n* sizeof(float));
```

Pri alokaciji dvodimenzionalne matrike velikosti $m \times n$ predpostavimo, da so vrstice urejene po vrsti v enodimenzionalno polje, za enostavnejši dostop do elementov pa običajno definiramo še vektor kazalcev, ki kažejo na začetke vrstic. Ustrezna alokacija dvodimenzionalne matrike A je določena z:

```
float **A, *Astorage;
int i;

Astorage = (float *)malloc(m*n*sizeof(float));
A = (float **)malloc(m*sizeof(float *));
for (i = 0; i < m; i++)
    A[i] = &Astorage[i*n];
```

```
#include <stdlib.h>
#include <mpi.h>

main(int argc, char** argv) {
    int numProcs, myRank;
    int i, j, m, n, blockSize = 3;
    float *X, *Y, *myY, *YY;
    float **A, *Astorage, **myA, **myAstorage;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);

    m = blockSize * numProcs; /* stevilo vrstic v A */
    n = 7; /* stevilo stolpcev v A */
    X = (float *) malloc(n*sizeof(float)); /* vektor X */
    if(myRank == 0) {
        Y = (float *) malloc(m*sizeof(float));  /* vektor Y */
        Astorage = (float *) malloc (m*n*sizeof(float)); /* mat. A */
        A = (float **)malloc(m*sizeof(float *));
        for(i = 0; i < m; i++) A[i] = &Astorage[i*n]; }
    else Astorage = A = Y = NULL;

    MPI_Bcast(X, n, MPI_FLOAT, 0, MPI_COMM_WORLD); /* oddamo  X */

    myY = (float *) malloc(blockSize*sizeof(float)); /* lokalno */
    myAstorage = (float *) malloc (blockSize*n*sizeof(float));
    myA = (float **)malloc(blockSize*sizeof(float *));
    for(i = 0; i < blockSize; i++) myA[i] = &myAstorage[i*n];

    MPI_Scatter(Astorage, blockSize*n, MPI_FLOAT, myAstorage,
                blockSize*n, MPI_FLOAT, 0, MPI_COMM_WORLD);

    for(i = 0; i < blockSize; i++)  /* lokalni izracun */
        {myY[i] = 0; for(j = 0; j < n; j++)
            myY[i] = myY[i] + myA[i][j] * X[j]; }

    MPI_Gather(myY, blockSize, MPI_FLOAT, Y,
               blockSize, MPI_FLOAT, 0, MPI_COMM_WORLD);

    if(myRank == 0) /* proces 0 izpise rezultat */
        for(i=0; i<m; i++) printf("\%f\n", Y[i]);

    free(X); free(Y); free(A); free(Astorage); /* brisanje */
    free(myY); free(myA); free(myAstorage);

    MPI_Finalize();
}
```

Slika 6.6: Množenje matrike in vektorja s funkcijami MPI za kolektivno komuniciranje.

6.4 Funkcije `MPI_Wtime`, `MPI_Wtick` in `MPI_Barrier`

Ko izdelamo paralelni program, nas seveda zanima, koliko smo na ta način pridobili. V ta namen lahko s pridom uporabimo funkcije

- `MPI_Wtime`,
- `MPI_Wtick` in
- `MPI_Barrier`.

Eden od načinov za oceno zmogljivosti paralelnega programa je merjenje časa izvajanja. Pri tem nas običajno ne zanimajo inicializacija MPI procesov, vzpostavitev komunikacije (čas latence) in čas za vhodno-izhodne operacije. Zanima pa nas pohitritev računanja zaradi povečanja števila procesorjev/računalnikov. Prva od zgoraj naštetih funkcij, `MPI_Wtime`, meri čas v sekundah od nekega dogodka v preteklosti, funkcija `MPI_Wtick` pa vrne natančnost rezultata funkcije `MPI_Wtime`, to je periodo, s katero se vrednost funkcije `MPI_Wtime` povečuje. Na primer, če funkcija `MPI_Wtick` vrne vrednost 10^{-6}, to pomeni, da se funkcija `MPI_Wtime` povečuje vsako mikrosekundo.

Sintaksa:
```
double MPI_Wtime(void);
double MPI_Wtick(void);
```

Če želimo časovno ovrednotiti določen del programske kode, moramo na njegov začetek in konec dodati klic funkcije `MPI_Wtime`. Razlika med vrednostima, ki jih vrne funkcija, je ravno čas izvajanja izbranega dela programske kode. Čeprav naj bi se vsak proces MPI začel izvajati v istem času, pa v praksi to ni res. Procesi MPI na različnih računalnikih se lahko začnejo izvajati tudi do več sekund narazen, kar seveda pomeni tudi različne čase za iste dele kode. Ta problem lahko rešimo s pomočjo funkcije `MPI_Barrier`, ki zagotavlja, da vsi procesi vstopajo v merjeno področje programske kode v istem času.

Sintaksa: `int MPI_Barrier (MPI_Comm comm);`

Parameter `comm` določa komunikator, katerega procesi so udeleženi v postopku sinhronizacije.

Primer, v katerem so uporabljene vse tri funkcije, je prikazan na sliki 6.7.

```
double start_time, end_time, elapsed_time;
MPI_Init (&argc, &argv);

MPI_Barrier (MPI_COMM_WORLD);

start_time = MPI_Wtime();
/* merjeni del programske kode */
end_time = MPI_Wtime();

elapsed_time = end_time - start_time;
```

Slika 6.7: Merjenje časa izvajanja paralelnega programa.

6.5 Primer programiranja z MPI: Eratostenovo sito

Algoritem za iskanje praštevil se imenuje po avtorju, grškem matematiku, ki je živel v času 276 - 194, pr. n. š. Preprosta psevdo-koda algoritma je naslednja:

1. Pripravi listo naravnih števil $2, 3, \ldots, n$ tako, da vsako število spremlja binarni zaznamek, ki je na začetku za vsa števila postavljen na 0.

2. Postavi spremenljivko k na prvo neoznačeno število v listi števil. Na začetku je to 2.

3. Ponavljaj, dokler je $k^2 < n$

   (a) Označi vse mnogokratnike števila k med k^2 in n.

   (b) Poišči najmanjše število, večje od k, ki ni označeno, in postavi k na to vrednost.

4. Neoznačena števila so praštevila.

Naš program bo kot rezultat vrnil število praštevil, manjših od števila n.

Prikazani algoritem ni praktičen za velike vrednosti n, saj je njegova kompleksnost enaka $O(n \ln(\ln n))$. Zanima nas paralelna oblika algoritma. Ker bomo uporabljali programski jezik C s funkcijami MPI, je priročno, da za predstavitev naravnih števil $2, 3, \ldots, n$ uporabimo polje $n-1$ znakov z indeksi $0, 1, \ldots, n-2$. Vrednost znaka pri indeksu i označuje, ali je naravno število $i+2$ označeno (znak je 1), ali ne (znak je 0).

6.5.1 Izvori paralelnosti

Z najbolj fino delitvijo na vsak element polja znakov vežemo osnovno opravilo, kar pomeni, da imamo $n-1$ osnovnih opravil (procesov). Ključni korak paralelnega računanja je korak 3(a) algoritma. V primeru opravila j to pomeni računanje mnogokratnikov števila k oziroma pogoja $j \bmod k = 0$, pri katerem je potrebno število j označiti. V tem primeru je v koraku 3(b) potrebna operacija redukcije (`MPI_Reduce`) za določitev nove vrednosti k in nato razpošiljanje (`MPI_Bcast`) te vrednosti vsem procesom.

6.5.2 Podatkovna delitev

V primeru Eratostenovega sita sta možni dve podatkovni dekompoziciji:

- delitev s prepletanjem in
- delitev na bloke.

Delitev s prepletanjem

V tem primeru je proces 0 odgovoren za naravna števila: $2, 2+p, 2+2p, \ldots$, proces 1 za $3, 3+p, 3+2p, \ldots$, in tako dalje. Prednost te delitve je, da indeks polja i enostavno določa proces, ki polje kontrolira ($i \bmod p$), njena slabost pri naši aplikaciji pa, da vodi v neenakomerno obremenjene procese. Primer delitve s prepletanjem prikazuje tabela 6.1. Na primer, za število 9, ki ima v polju znakov indeks $j = 7$, je zadolžen

Tabela 6.1: Delitev s prepletanjem na primeru Eratostenovega sita za $n = 15$, $p = 4$.

Proces	0				1				2			3		
Število	2	6	10	14	3	7	11	15	4	8	12	5	9	13
Indeks	0	4	8	12	1	5	9	13	2	6	10	3	7	11

proces $7 \bmod 4 = 3$.

Delitev na bloke

Ta delitev deli polje v p stičnih blokov približno enakih dolžin. Če je dolžina polja n mnogokratnik števila p, so bloki enako veliki. Poznani sta dve delitvi na bloke: strnjena in razpršena.

Strnjena delitev na bloke Pri dolžini polja m in številu procesov p naslednji izrazi določajo:

- prvi element v polju procesa i: $i\lfloor m/p \rfloor + \min(i, r)$,
- zadnji element v polju procesa i: $(i+1)\lfloor m/p \rfloor + \min(i+1, r) - 1$,
- j-ti element pripada procesu: $\min\left(\lfloor j/\left(\lfloor m/p \rfloor + 1\right)\rfloor, \lfloor (j-r)/\lfloor m/p \rfloor \rfloor\right)$, kjer je $r = m \bmod p$ in $\lfloor\ \rfloor$ zaokrožitev navzdol.

O strnjeni delitvi govorimo zato, ker bloki z enakim številom elementov stojijo eden ob drugem.

Primer strnjene delitve na bloke za $n = 15$ $(m = 14), p = 4$ prikazuje tabela 6.2. Prvi in zadnji element v polju procesa sta izračunana po zgornjih enačbah. Število

Tabela 6.2: Strnjena delitev na bloke za $n = 15$ $(m = 14), p = 4$.

Proces	0				1				2			3		
Število	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13

9, ki ima v polju znakov indeks $j = 7$, je pri tej delitvi vsebovano v procesu, ki ga določa izraz: $\min(\lfloor 7/(3+1) \rfloor, \lfloor (7-2)/3 \rfloor) = \min(1, 1) = 1$.

Razpršena delitev na bloke V tem primeru so bloki z različnim številom elementov pomešani med seboj. Pri dolžini polja m in številu procesov p naslednji izrazi določajo:

- prvi element v polju procesa i: $\lfloor im/p \rfloor$,
- zadnji element v polju procesa i: $\lfloor (i+1)m/p \rfloor - 1$,
- j-ti element pripada procesu: $\lfloor (p(j+1) - 1)/n \rfloor$.

Primer razpršene delitve na bloke za $n = 15$ $(m = 14), p = 4$ prikazuje tabela 6.3. Prvi in zadnji element v polju procesa sta izračunana po zgornjih enačbah. Število

Tabela 6.3: Strnjena delitev na bloke za $n = 15$ $(m = 14), p = 4$.

Proces	0			1				2			3			
Število	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13

9, ki ima v polju znakov indeks $j = 7$, je pri tej delitvi vsebovano v procesu, ki ga določa izraz: $\lfloor (4(7+1)-1)/14 \rfloor = 2$.

Pri strnjeni delitvi na bloke smo dobili štiri bloke (procese), prva dva s po štirimi elementi in druga dva s po tremi, pri razpršeni delitvi pa smo dobili bloke s po 3, 4, 3 in 4 elementi. Razpršena delitev na bloke je boljša, saj so izračuni enostavnejši in zato hitrejši.

V programski kodi (podana je v dodatku A) so na osnovi zgornjih izrazov za prvi element v bloku, zadnji element v bloku, velikost bloka in pripadnost indeksa procesu za razpršeno delitev na bloke, uporabljeni naslednji makroji jezika C:

- `BLOCK_LOW(rank, p, m)`,
- `BLOCK_HIGH(rank, p, m)`,
- `BLOCK_SIZE(rank, p, m)` in
- `BLOCK_OWNER(j, p, m)`.

6.5.3 Razvoj paralelnega programa

Pri delitvi podatkovnega polja v bloke moramo ločevati med lokalnim indeksom znotraj posameznega bloka in globalnim indeksom, ki teče preko vseh blokov. Razliko med lokalnimi in globalnimi indeksi prikazuje tabela 6.4.

Tabela 6.4: Lokalni in globalni indeksi elementov v blokih.

Proces	0			1				2			3			
Lokalni indeks	0	1	2	0	1	2	3	0	1	2	0	1	2	3
Globalni indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Pri delitvi na bloke pri primeru Eratostenovega sita je potrebno vedeti, da je največje praštevilo, ki se uporablja v algoritmu za označevanje števil med k^2 in n, enako $\sqrt{n}$. Če je v prvem bloku $n/p > \sqrt{n}$ podatkov, potem bo prvi procesor določil vsa praštevila za označevanje in ne bo potrebne komunikacije za njihovo določitev.

Naslednja prednost delitve na bloke je v pohitritvi označevanja celic, ki so mnogokratniki števila k. Namesto n/p operacij modulo za vsako praštevilo, lahko določimo mnogokratnike mnogo hitreje tako, da poiščemo prvega, na primer z oznako j, nato pa sledi označitev po vrsti $j, j+k, j+2k, \ldots$ do konca bloka.

Sedaj, ko smo določili delitev podatkov v bloke, si oglejmo, kako izgleda paralelna izvedba Eratostenovega sita. Po vrsti si oglejmo posamezne korake, kot sledijo iz algoritma, podanega na začetku poglavja.

V prvem koraku vsak od p procesov pripravi del polja, ki vsebuje ali $\lfloor n/p \rfloor$ ali $\lceil n/p \rceil$ znakovnih polj. Vsak proces potrebuje število k, da lahko označuje mnogokratnike števil v svojem delu polja (bloku). Če privzamemo, da je $p < \sqrt{n}$ oziroma $n/p > \sqrt{n}$, je izključno prvi proces (proces 0) odgovoren za določitev naslednje vrednosti k, ki jo razpošlje vsem ostalim procesom. Korak 3(a) zahteva za vsak proces določitev prvega mnogokratnika števila k v bloku. To dosežemo s pomočjo makroja `BLOCK_LOW`. Temu sledi označevanje mnogokratnikov števila k v bloku od k^2 do n. Nato proces 0 ponovno razpošlje naslednjo vrednost števila k vsem ostalim procesom. Ko je $k^2 > n$ in so vsa števila označena, vsak proces prešteje število praštevil oziroma neoznačenih znakov v svojem bloku in jih pošlje procesu 0, ki jih sešteje in izpiše.

Analiza paralelnega Eratostenovega algoritma

Vzemimo, da χ predstavlja čas, potreben za označevanje ene celice, čas za povečanje indeksa zanke in testiranje zaključnega pogoja zanke. Določimo ga lahko eksperimentalno. Iz časovne kompleksnosti zaporednega algoritma $O(n \ln \ln n)$ lahko ocenimo pričakovani čas izvajanja zaporednega programa kot $\chi n \ln \ln n$.

V vsaki iteraciji algoritma se prenaša samo število k. Čas potreben za prenos je približno enak $\lambda \lceil \log_2 p \rceil$, kjer je λ latenca. Ker je med 2 in n približno $n/\ln n$ praštevil, je dobra aproksimacija za število iteracij enaka $\sqrt{n}/\ln \sqrt{n}$. Torej je pričakovan čas izvajanja paralelnega algoritma približno enak: $\chi n \ln \ln n/p + \lambda \lceil \log_2 p \rceil \sqrt{n}/\ln \sqrt{n}$.

Možnih je precej izboljšav predlaganega paralelnega algoritma:

1. Ker razen števila 2 nobeno praštevilo ni sodo, lahko listo števil skrčimo na polovico, kar zmanjša potreben pomnilniški prostor in podvoji hitrost označevanja mnogokratnikov. S tem postane pričakovan čas izvajanja zaporednega programa enak $\chi n \ln \ln n/2$, ocenjeni čas izvajanja paralelnega programa pa enak $\chi n \ln \ln n/(2p) + \lambda \lceil \log_2 p \rceil \sqrt{n}/\ln \sqrt{n}$.
2. Če vsakemu procesu dodamo blok podatkov $2, 3, \ldots, \sqrt{n}$, zaradi replikacije sicer izgubimo nekaj pomnilniškega prostora, vendar s to spremembo lahko vsak proces sam določa novo vrednost k v vsakem koraku iteracije. S tem je odpadla potreba, da proces 0 pošilja vrednosti k ostalim procesorjem. Pričakovan čas izvajanja paralelnega programa se s tem zmanjša na $\chi n \ln \ln n/(2p) + \sqrt{n} \ln \ln \sqrt{n} + \lambda \lceil \log_2 p \rceil$.
3. Zanko `repeat ... until` lahko zapišemo kot dve zanki, zunanjo, ki gre preko praštevil v območju $3, \ldots, \lfloor \sqrt{n} \rfloor$ in notranjo, ki iterira preko celotne liste

$3, \ldots, n$. Če zanki zamenjamo, močno povečamo verjetnost za zadetek v predpomnilniku procesorja (*ang.* cache hit rate). Paziti moramo le, da velikost polja (bloka), do katerega dostopamo v notranji zanki, ne presega velikosti strani v predpomnilniku. Tedaj v naslednjem koraku zunanje zanke pride do zgrešitve v predpomnilniku in s tem do vpisa novih vrednosti.

V dodatku A.1 je podana kompletna programska koda v jeziku *C* za paralelno različico programa Eratostenovo sito.

6.6 Primer programiranja z MPI: Floydov algoritem

Floydov algoritem predstavlja klasično metodo za določitev najkrajše razdalje med vsemi pari v uteženem usmerjenem grafu. Razdalja med dvema vozliščema je podana kot utež povezave. Utežen usmerjen graf lahko predstavimo s sosednostno matriko, v kateri element (i, j) predstavlja razdaljo od vozlišča i do vozlišča j. Za neobstoječe povezave vnesemo neskončno razdaljo.

Vzemimo, da je vhod v Floydov algoritem število vozlišč v grafu n, skupaj z ustrezno sosednostno matriko $A_{0\ldots n-1 \times 0\ldots n-1}$. Izhod algoritma je preoblikovana matrika $A_{0\ldots n-1 \times 0\ldots n-1}$, v kateri so vpisane najkrajše razdalje med pari vozlišč. Psevdo koda Floydovega algoritma izgleda takole

```
for k = 0 to n-1
    for i = 0 to n-1
        for j = 0 to n-1
            a[i, j] = min(a[i, j], a[i, k] + a[k, j])
        end for
    end for
end for
```

Razvoj paralelnega algoritma

Delitev V tem koraku se odločamo za podatkovno ali funkcijsko delitev. V tem primeru je odločitev preprosta, saj iz psevdo-kode izhaja, da se operacija prireditve izvaja n^3-krat. Na dlani je podatkovna dekompozicija. Vsak element sosednostne matrike je osnovna celica z ustreznim osnovnim opravilom. Torej izberemo v tem koraku delitev sosednostne matrike na n^2 elementov in enako število osnovnih opravil.

Komunikacija Vsaka posodobitev elementa `a[i,j]` zahteva dostop do elementov `a[i,k]` in `a[k,j]`. To pomeni, da se med k-to iteracijo vsak element v vrstici k pošlje (*ang.* broadcast) vsem opravilom v istem stolpcu (j). Podobno se vsak element v stolpcu k pošlje vsem opravilom v isti vrstici (i). Naslednje vprašanje je, ali lahko vse elemente matrike posodobimo paralelno? Odgovor je pritrdilen, saj se `a[i,k]` in `a[k,j]` ne spreminjata med k-to iteracijo. Tedaj namreč posodobitev elementa `a[i,k]` izgleda takole:

```
a[i,k] = min(a[i,k], a[i,k] + a[k,j]),
```

kjer so vse vrednosti pozitivne, zato se `a[i,k]` ne more zmanjšati. Podobno velja za posodobitev `a[k,j]`:

```
a[k,j] = min(a[k,j], a[k,k] + a[k,j]),
```

zato se tudi `a[k,j]` ne more zmanjševati. Torej, posodobitev elementa `a[i,j]` ni odvisna od posodobitve elementov `a[i,k]` in `a[k,j]`. Zato se lahko v vsaki iteraciji k (zunanja zanka) elementi vrstice k najprej razpošljejo vsem ostalim elementom, nato pa sledi paralelno posodabljanje elementov matrike A.

Združevanje in preslikava Cilj združevanja je zmanjšati komunikacijo tako, da n^2 osnovnih opravil želimo združiti v p sestavljenih opravil. Imamo dve možnosti: da združujemo opravila v isti vrstici ali v istem stolpcu sosednostne matrike. Tedaj bo med vsako zunanjo iteracijo k vsako osnovno opravilo oddajalo n elementov vsem ostalim opravilom. Vsaka oddaja pri tem zahteva $\lceil \log_2 p \rceil (\lambda + n/\beta)$ časa. Zaradi enostavnejše uporabe funkcije `MPI_Bcast` ima prednost združevanje elementov v okviru vrstic.

Najprej si oglejmo branje matrike razdalj iz datoteke. Predpostavimo, da bo le en proces odgovoren za branje matrike. Ta proces bo prebral matriko in jo porazdelil po ostalih procesih. V primeru p procesov je najbolje, da proces $p-1$ prebere matriko, saj je ta proces odgovoren za branje $\lceil n/p \rceil$ vrstic, kar je največ med vsemi procesi. Zato je najprimernejši tudi za začasno shranjevanje in izpis blokov vrstic iz ostalih procesorjev.

Varianta Floydovega paralelnega algoritma, katerega koda je podana v dodatku A.2, izpiše najprej začetno sosednostno matriko, ob koncu pa še popravljeno sosednostno matriko z najkrajšimi razdaljami med vozlišči. Proces $p-1$ upravlja z vhodom in izhodom ter vodi komunikacijo z ostalimi procesi, ostali procesi pa čakajo na svoje vrstice in po izračunu njihovih najkrajših razdalj pošljejo vrstice nazaj procesu $p-1$.

6.6.1 Paralelna izvedba Floydovega algoritma

V dodatku A.2 je podana kompletna paralelna koda Floydovega algoritma.

Funkcija `readAndDistributeMatrix` ob podanem imenu vhodne datoteke, v kateri je zapisana matrika, vrne naslov polja kazalcev na vrstice matrike, kazalec na pomnilnik z matričnimi vrednostmi in dimenzije matrike.

Funkcija `computeShortestPaths` ima 4 parametre: rang procesorja, število procesorjev, kazalec na procesorjev blok vrstic iz matrike razdalj in velikost matrike. Med vsako iteracijo (vrednostjo k) se k-ta vrstica porazdeli od procesa, ki hrani to vrstico, do vseh ostalih s kolektivno oddajo (*ang.* broadcast). Temu sledi paralelno računanje novih razdalj v vseh procesorjih za vrstice, ki so jim določene z vrstično delitvijo. Vsak proces pri tem rezervira polje celih števil z imenom `Ak`, ki shranjuje k-to vrstico, v vsakem koraku iteracije drugo.

Ob koncu zunanje zanke klic funkcije `printDistributedMatrix` v procesu $p-1$ izpiše rezultat.

6.6.2 Analiza paralelnega algoritma

Iz psevdo kode Floydovega algoritma sledi, da je kompleksnost zaporedne različice enaka $O(n^3)$. Očitno notranja zanka, ki posodablja elemente ene vrstice v matriki A, ustreza kompleksnosti $O(n)$. V primeru združevanja vrstic znotraj sestavljenega opravila (procesa) vsak takšen proces izvede največ $\lceil n/p \rceil$ iteracij srednje zanke, zato ima kompleksnost $O(n^2/p)$. Pred srednjo zanko je oddaja `MPI_Bcast`, ki prenese vrstico matrike vsem procesom. Prenos enemu procesu ima kompleksnost $O(n)$, prenos p procesorjem pa zahteva $\lceil \log_2 p \rceil$ korakov, oziroma kompleksnost operacije oddaje v vsaki iteraciji je $O(n \log_2 p)$.

Vsaka iteracija zunanje zanke paralelnega algoritma zahteva računanje novega korenskega procesa, kar vzame konstanten čas. Ta nato kopira pravilno vrstico matrike v polje `tmp`, kar vzame $O(n)$ časa. Zunanja zanka se izvede n-krat. Časovna kompleksnost paralelnega algoritma je tako ocenjena na: $O(n(1+n+n\log_2 p+n^2/p)) = O(n^3/p + n^2 \log_2 p)$.

V primeru mreže komercialnih računalnikov traja vsaka oddaja $\lceil \log_2 p \rceil$ korakov. Vsak korak pomeni prenos sporočila, dolgega $4n$ bajtov (podatkovni tip integer zavzame v pomnilniku 4 bajte). V tem primeru je pričakovani čas komunikacije enak $n\lceil \log_2 p \rceil(\lambda + 4n/\beta)$. Če je χ povprečen čas, potreben za posodobitev enega elementa matrike, je pričakovan čas računanja paralelnega programa enak $n^2\lceil n/p \rceil \chi$. Če

dodamo še čas oddaje, dobimo končno oceno časa izvajanja paralelnega algoritma:

$$t_{\text{exe}} = n^2\lceil n/p\rceil\chi + n\lceil\log_2 p\rceil(\lambda + 4n/\beta) \quad . \tag{6.1}$$

Ta ocena pa je pretirana, saj ne upošteva prekrivanja med komunikacijo in računanjem. V primeru, da čas računanja na iteracijo presega čas za prenos sporočila, lahko vsak proces porabi enak čas za sprejem ali oddajo sporočila, ki je enak $\lceil\log_2 p\rceil\lambda$. Če je $\lceil\log_2 p\rceil 4n/\beta < n\lceil n/p\rceil\chi$, potem je čas prenosa sporočila po prvi iteraciji popolnoma prekrit s časom računanja, zato ga lahko zanemarimo:

$$t_{\text{exe}} = n^2\lceil n/p\rceil\chi + n\lceil\log_2 p\rceil\lambda + \lceil\log_2 p\rceil 4n/\beta) \quad . \tag{6.2}$$

Poglavje 7

Programiranje s knjižnico OpenMP

V prejšnjem poglavju je bil opisan način paralelnega programiranja z uporabo knjižnice MPI. V primeru, da imajo procesorji skupen pomnilnik, pa je mogoče pogosto priti do boljših lastnosti (večje učinkovitosti) s programskim standardom OpenMP, ki predstavlja uporabniški programski vmesnik (*ang.* Application Programming Interface) za paralelno programiranje večprocesorskih sistemov [16, 21]. Sestoji se iz niza ukazov za prevajalnik in knjižnice podpornih funkcij. OpenMP deluje v kombinaciji s standardnimi jeziki kot so Fortran, *C* ali *C++*. Podobno kot pri knjižnici MPI, potrebujemo na začetku programa deklaracijo knjižnice `#include <omp.h>`.

7.1 Model deljenega pomnilnika

Model deljenega pomnilnika je abstrakcija generičnega centraliziranega večprocesorskega sistema, kot je bil definiran v poglavju o paralelnih arhitekturah. Gre za sistem z nizom procesorjev, ki so povezani preko skupnega pomnilnika. Procesorji v tem primeru lahko komunicirajo in se sinhronizirajo med seboj preko deljenih spremenljivk.

Standardni pogled na paralelnost v modelu deljenega pomnilnika je tako imenovani paralelizem tipa razdruži/združi (*ang.* fork/join). Ob začetku programa obstaja samo ena programska nit (*ang.* thread), ki se imenuje glavna nit (*ang.* master thread) in izvaja sekvenčni del algoritma. V točkah, kjer so mogoče paralelne operacije, se glavna nit razdruži (*ang.* fork) na dodatne niti, ki se izvajajo vzporedno z glavno nitjo. Ko so paralelne operacije končane, se dodatne niti ponovno združijo (*ang.* join) v glavno nit. Takšno razdruževanje/združevanje se v algoritmu lahko ponovi poljubno mnogokrat.

Glavna razlika med modelom MPI in modelom OpenMP je, da pri modelu MPI vsi procesi tipično ostanejo aktivni skozi ves čas izvajanja programa, medtem ko se pri modelu OpenMP število aktivnih niti dinamično spreminja med izvajanjem programa.

Zaporedni program z eno samo nitjo je poseben primer paralelnega programa za arhitekturo deljenega pomnilnika. Model deljenega pomnilnika podpira postopno paralelizacijo zaporednega algoritma. Preoblikovanje algoritma poteka po korakih, tako, da se v vsakem koraku preoblikuje po en del kode. Na podlagi ocene časov izvajanja posameznih delov zaporednega algoritma se lahko odločimo samo za paralelizacijo najbolj časovno kritičnih delov, po potrebi pa nato nadaljujemo še s časovno manj zahtevnimi deli, vse dokler nam izboljšave opravičujejo vloženo delo. Postopna paralelizacija, ki jo omogoča model OpenMP, je ena od pomembnih prednosti pred modelom MPI, pri katerem paralelizacija zahteva velike spremembe v zasnovi programske kode.

7.1.1 Ukazi prevajalniku in njihova dopolnila

Paralelne operacije so v zaporednih programih pogosto izražene z zankami. Z ukazi prevajalniku, ki jih ponuja knjižnica OpenMP, lahko enostavno označimo, kdaj je iteracije v zanki mogoče izvajati paralelno. To pomeni, da je potrebno posebej označiti zanke, ki zaradi narave iteracij (aplikacije) omogočajo paralelno izvajanje. V ta namen je mogoče uporabiti posebne ukaze prevajalniku, ki so v jeziku *C* imenovani pragma (*ang.* pragmatic information). Informacija, ki jo prevajalniku daje tak ukaz, za prevajalnik ni obvezna in jo lahko ignorira ter kljub temu generira korektno izvršilno datoteko. V primeru, da prevajalnik tak ukaz upošteva, pa lahko zgenerira bolj optimalno izvršilno datoteko.

Ukaz prevajalniku v jezikih *C* ali *C++* ima sintakso:

```
#pragma omp <ukaz in dopolnila>
```

Prvi ukaz prevajalniku, ki jo bomo spoznali, je `parallel for` . Njegova sintaksa je:

```
#pragma omp parallel for
```

Če to vrstico napišemo tik nad zanko, od prevajalnika zahtevamo, da poskusi zanko paralelizirati.

```
#pragma omp parallel for
for (i = first; i < size; i += prime)
    marked[i] = 1;
```

Za uspešno prevedbo zanke v zaporednem programu v paralelno različico mora prevajalnik preveriti, če so v času izvajanja programa vse informacije, potrebne za določitev števila iteracij na osnovi kontrolnega stavka, dostopne. Zato mora imeti kontrolni stavek kanonično obliko - podane morajo biti vse možne variante kontrolnega stavka. Poleg tega zanka ne sme vsebovati stavka za predčasni izhod iz zanke (na primer, `break`, `return`, `exit`, `goto`).

Med paralelnim izvajanjem zanke glavna nit kreira dodatne niti tako, da vse skupaj pokrivajo iteracije znotraj zanke. Vsaka nit ima svoj izvršilni kontekst oziroma naslovni prostor vseh spremenljivk, do katerih lahko dostopa. Pri tem imajo deljene spremenljivke (*ang.* shared variables) isti naslov v izvršilnem kontekstu za vse niti, privatne spremenljivke (*ang.* private variables) pa imajo različne naslove v izvršilnem kontekstu vsake niti. Pri ukazu prevajalniku `parallel for` so vse spremenljivke praviloma deljene, razen indeksa zanke, ki je privatna spremenljivka.

Vprašanje, ki se seveda vsiljuje, je, koliko niti naj kreira prevajalnik na osnovi ukaza. V ta namen obstaja sistemska spremenljivka `OMP_NUM_THREADS`, ki določa predvideno število niti in se običajno ujema s številom procesorjev v sistemu. Število niti se lahko s klicem ustrezne funkcije tudi programsko spremeni.

Funkcija

```
int omp_get_num_procs(void)
```

vrne število fizičnih procesorjev, ki so na voljo paralelnemu programu. Vrnjeno celo število je lahko manjše od celotnega števila procesorjev v sistemu, odvisno od strategije sistema v času izvajanja programa.

Funkcija

```
void omp_set_num_threads(int t)
```

s parametrom `t` postavi število aktivnih niti v paralelnem delu programske kode. To funkcijo lahko v programu kličemo večkrat in na ta način vplivamo na stopnjo paralelnosti. Naslednji primer prikazuje, kako lahko določimo število niti glede na število dosegljivih procesorjev:

```
int t;

t = omp_get_num_procs ();
omp_set_num_threads (t);
```

7.2 Povečevanje učinkovitosti

V primeru vgnezdenih zank lahko vplivamo na zrnatost ali granulacijo (*ang.* grain size) paralelnih delov programske kode. Če paralelno izvedemo zunanjo zanko, povečamo stopnjo zrnavosti, če pa se omejimo le na notranjo zanko, potem stopnjo zrnavosti v paralelnem programu zmanjšamo. V splošnem velja, da s povečanjem zrnavosti izboljšamo lastnost paralelnega programa. V primeru paralelnega izvajanja zunanje zanke je pomembno, da so indeksi notranjih zank privatne spremenljivke. V nasprotnem primeru bi se lahko zgodilo, da se ne bi izvedle vse notranje iteracije, saj bi paralelne niti poizkušale inicializirati in povečevati isto deljeno (indeksno) spremenljivko.

Lahko se zgodi, da preoblikovanje zanke v zaporednem algoritmu v paralelno različico poveča čas izvajanja programa. Zato je potrebno izboljšati lastnosti paralelnih zank. V nadaljevanju si bomo ogledali tri načine.

Obrat zank

Vzemimo naslednji del programa

```
for (i = 1; i < m; i++)
    for (j = 0; j < n; j++)
        a[i][j] = 2 * a[i-1][j];
```

Zgornja koda ne dopušča preračunavanja elementov v dveh vrsticah hkrati, saj so vrstice med seboj podatkovno odvisne. Pač pa je mogoče hkrati preračunavati hkrati elemente v stolpcih. To pomeni, da lahko zanko z indeksom `j` izvedemo paralelno, zanke z indeksom `i` pa ne. Z uporabo ukaza prevajalniku `parallel for` pred notranjo zanko ne dobimo dobrega rezultata, ker bi z njo dosegli $m - 1$ korakov razdruži/združi (fork/join), po enega na iteracijo zunanje zanke. Če pa zanki obrnemo, je potreben le en korak razdruži/združi, ki obdaja zunanjo zanko. Izboljšana koda izgleda takole:

```
#pragma parallel for private (j)
for (j = 0; j < n; j++)
    for (i = 1; i < m; i++)
        a[i][j] = 2 * a[i-1][j];
```

Pri zgornjem obratu zanke moramo biti pozorni tudi na to, kako le-ta vpliva na verjetnosti za zadetke v predpomnilniku (*ang.* cache hit rate). Vsaka nit namreč

po obratu deluje nad stolpci matrike namesto nad vrsticami, kar v primeru zapisa matrike v pomnilniku po vrsticah poslabša izrabo predpomnilnika. Dopolnilo `private`, ki sledi ukazu `parallel for`, pomeni, da niti spremenljivko v oklepaju lahko vzamejo za privatno.

Pogojno izvajanje zank

Če zanka nima dovolj iteracij, potem je čas za izvedbo koraka razdruži/združi prevelik, da bi opravičil sočasno izvedbo zanke. Tedaj lahko z dopolnilom `if` določimo pogoj, pri katerem se bo zanka izvajala paralelno. Pogoj ima sintakso `if (<skalarni izraz>)`. Primer:

```
#pragma omp parallel for private (x) if (n > 5000)
```

V tem primeru se zanka, ki sledi, izvede paralelno le, če je izpolnjen pogoj, v našem primeru $n > 5000$.

Razvrstitev iteracij po nitih

Včasih se v zankah čas izvajanja različnih iteracij precej razlikuje. Z dopolnilom `schedule` lahko določimo, kako razvrstiti posamezne iteracije po nitih. V primeru statičnega razvrščanja (*ang.* static schedule) se vse iteracije dodelijo nitim, še preden jih le-te izvršijo. Pri dinamičnem razvrščanju (*ang.* dynamic schedule) pa se le nekaj iteracij dodeli nitim na začetku izvajanja zanke. Niti, ki končajo svoje iteracije, so proste in dostopne za novo delo. Dodelitev iteracij se tako nadaljujejo do konca zanke. Za statično razvrščanje je značilen nizek presežek (*ang.* overhead) in visoka neuravnoteženost (*ang.* imbalance), dinamično razvrščanje pa ima višji presežek, toda zmanjšano neuravnoteženost. V obeh primerih se preslika podoben obseg iteracij na niti. Na obseg lahko tudi vplivamo preko parametra. S povečanjem obsega se zmanjša presežek in poveča izraba predpomnilnika. Z njegovim zmanjšanjem pa se lahko izboljša uravnoteženje bremena po nitih. Sintaksa je naslednja:

```
schedule (<tip>[,<obseg>])
```

Parameter `tip` je potrebno podati, parameter `obseg` pa po potrebi. S parametroma `tip` in `obseg` je mogoče opisati veliko različnih načinov dodeljevanja iteracij po nitih. Primeri:

- `schedule (static)`: statična dodelitev iteracij na nit. Vsaka nit dobi `n/t` iteracij, kjer je `n` število vseh iteracij, `t` pa število niti,

- `schedule (static, C)`: prepletena dodelitev, `C` je obseg iteracij na nit,
- `schedule (dynamic)`: dinamično dodeljevanje iteracij (po eno naenkrat),
- `schedule (dynamic, C)`: dinamično dodeljevanje iteracij, po `C` naenkrat.

Dinamično dodeljevanje je mogoče programsko tudi vklapljati in izklapljati s pomočjo funkcij `omp_get_dynamic()` in `omp_set_dynamic()`. Primer:

```
int main()
{
    omp_set_dynamic (false);
    omp_set_num_threads (omp_get_num_procs() );
}
```

7.3 Deklaracije privatnih spremenljivk

Spremenljivke v primeru modela deljenega pomnilnika so običajno deljene. S posebnimi dopolnili k ukazom prevajalniku pa jih lahko spremenimo. Med takšna dopolnila sodijo `shared`, `private`, `firstprivate`, `lastprivate`, `threadprivate`.

Dopolnilo `private` usmerja prevajalnik tako, da ustvari eno ali več privatnih spremenljivk. Njegova sintaksa je naslednja:

```
private (<lista spremenljivk>)
```

To dopolnilo pove prevajalniku, naj vsaki niti dodeli privatno kopijo spremenljivke. Primer uporabe dopolnila `private` v kombinaciji z vgnezdeno `for` zanko:

```
#pragma omp parallel for private (j)
for (i = 0; i < BLOCK_SIZE (id, p, n); i++)
    for (i = 0; j < n; j++)
        a[i][j] = MIN (a[i][j], a[i][k] + Ak[j]);
```

V tem primeru je vrednost privatne spremenljivke nedefinirana pri prehodu na paralelni del kode, prav tako je njena vrednost nedefinirana ob izhodu iz paralelne kode. Včasih želimo, da privatna spremenljivka deduje vrednost deljene spremenljivke. Temu je namenjeno dopolnilo

```
firstprivate (<lista spremenljivk>),
```

ki usmeri prevajalnik tako, da kreira privatno spremenljivko z začetno vrednostjo enako vrednosti deljene spremenljivke, kontrolirane s strani glavne niti ob vhodu v zanko. Primer:

```
x[0]=complex_function();

#pragma omp parallel for private (j) firstprivate (x)
for (i=0;i<n;i++)
{
    for (j = 1; j < 4; j++)
        x[j] = g (i, x[j-1]);
    answer[i] = x[1] - x[3];
}
```

Zaradi uporabe dopolnila `firstprivate` se je začetna vrednost `x[0]` dedovala na privatne spremenljivke vsake niti notranje zanke. Ker takšna inicializacija velja le za prvo iteracijo vsake niti, ima parameter predpono first.

Podobno je vloga dopolnila `lastprivate`, da usmeri prevajalnik h generiranju kode ob koncu paralelizirane zanke tako, da kopira zadnjo vrednost privatne spremenljivke, ki se nahaja v niti zadnje iteracije zanke, v deljeno spremenljivko. Ukaz prevajalniku `parallel for` lahko vsebuje tako dopolnilo `firstprivate` kot tudi dopolnilo `lastprivate`. Pri tem imata dopolnili lahko nič, nekaj ali vse skupne spremenljivke.

Z dopolnilom `threadprivate` napravimo globalno spremenljivko za sestavni del vsake niti. To se razlikuje od dopolnila `private`, s katerim le maskiramo globalne spremenljivke. Z dopolnilom `threadprivate` dobi vsaka nit kopijo globalne spremenljivke, ki pa se ohrani med izvajanjem paralelnih blokov. Tudi te spremenljivke je mogoče inicializirati, in sicer z dopolnilom `copyin`.

7.4 Sinhronizacija

Standard OpenMP podpira naslednje ukaze za sinhronizacijo: `critical section`, `atomic`, `barrier`, `single`, `master`, `flush` in `ordered`. Zadnjih dveh v nadaljevanju ne bomo obravnavali.

Če so iteracije v zanki odvisne ena od druge, lahko paralelizacija takšne zanke v kombinaciji z uvedbo privatnih spremenljivk pripelje do napačnega rezultata. V tem primeru pride do tekmovalnih okoliščin (*ang.* race condition), ki v primeru, ko več niti uporablja deljeno spremenljivko, vodijo v nedeterministično računanje. Takšno, ali pa kakšno drugo kritično točko v programu lahko označimo z ukazom

`critical`, ki opozori prevajalnik, da naj poskrbi za medsebojno izključitev niti v označenem predelu kode. Sintaksa ukaza je:

```
#pragma omp critical (name),
```

kjer je `name` oznaka kritičnega odseka kode. Primer, ki z označitvijo kritične kode prepreči napako:

```
double area, pi, x;
int i, n;

area = 0.0;

#pragma omp parallel for private (x)
for (i = 0; i < n; i++)
{
    x = (i + 0.5) / n;
    #pragma omp critical
    area += 4.0 / (1.0 + x * x);
}
pi = area / n;
```

V tem primeru je zagotovljeno, da se bodo iteracije razdelile med niti tako, da bo samo ena nit naenkrat izvršila del kode, pred katerim je ukaz `critical`, v zgornjem primeru prireditveni stavek, ki spremeni vrednost spremenljivke `area`. S tem pa nismo dosegli nobene pohitritve.

Ukaz `atomic` je tip kritične sekcije, ki zagotavlja, da jo izvede le ena nit naenkrat. Uporabimo jo lahko edino za zagotovitev pravilnega izvajanja prireditvenih stavkov, ki imajo obliko $var_{op} = expr$, kjer je *op* lahko $+, -, *, /, \&,$ `<<` ali `>>`, `var op = expr` ali `++var`, `--var`, `var++` in `var--`. Primer:

```
int x, b;

x = 0;
#pragma omp parallel private (b)
{
    b = calculate (i);
    pragma omp atomic
    x += b;
}
```

Ukaz `barrier` zagotovi, da na točki ukaza niti čakajo, dokler vse niti ne dosežejo te točke v programski kodi. Primer:

```
int A[1000];

#pragma omp parallel
{
    big_calculation_1 (A);
    #pragma omp barrier
    big_calculation_2 (A);
}
```

Z ukazom `single` označimo blok, ki ga izvede le ena nit. Vse ostale niti čakajo, dokler le-ta ne konča dela. Primer:

```
#pragma omp parallel
{
    perform_calculationA (omp_get_thread_num ());

    #pragma omp single
    printf("Smo med izracunom ...\n");

    perform_calculationB( omp_get_thread_num() );
}
```

Ukaz `master` označuje strukturiran blok, ki ga izvaja le glavna nit. Ostale niti ta ukaz ignorirajo. Primer:

```
#pragma omp parallel
{
    #pragma omp master
    printf ("Zacenjamo glavni izracun ...\n");

    perform_calculation (omp_get_thread_num ());
}
```

7.5 Operacija redukcije

Operacijo redukcije lahko uporabimo hkrati z ukazom `parallel for`. Podati moramo operacijo redukcije (*ang.* reduce) in spremenljivko, na kateri naj se operacija

izvaja. Knjižnica OpenMP bo poskrbela za detajle, kot so shranjevanje delnih vsot v privatnih spremenljivkah in dodajanje delnih vsot deljeni spremenljivki ob koncu zanke. Sintaksa dopolnila za operacijo redukcije je

```
reduction (<operacija>:<spremenljivka>)
```

V spodnjem primeru je prikazana uporaba dopolnila `reduction` pri numeričnem izračunu integrala $\int_0^1 \frac{4}{1+x^2} = \pi$ po metodi seštevanja pravokotnikov.

```
double area, pi, x;
int i, n;

area = 0.0;

#pragma omp parallel for private (x) reduction (+:area)
for (i = 0; i < n; i++)
{
    x = (i + 0.5) / n;
    area += 4.0 / (1.0 + x * x);
}
pi = area / n;
```

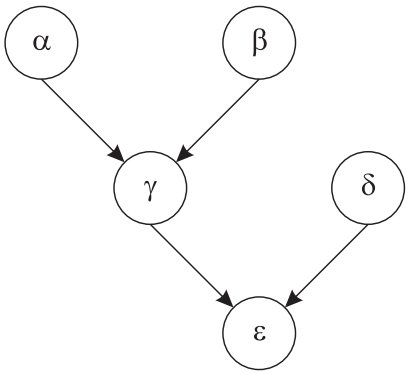
Z uporabo dopolnila `reduction` v ukazu `omp parallel for` smo nadomestili uporabo ukaza `critical` za označitev kritičnega dela programa. Izkaže se, da je čas izvajanja programa, v katerem uporabljamo operacijo redukcije, precej krajši od programa, v katerem označimo kritični del kode.

7.6 Funkcijska sočasnost

Do sedaj smo v okviru paralelnega programiranja s knjižnico OpenMP obravnavali le podatkovno sočasnost. Kot smo videli že pred tem, pa obstaja za paralelnost tudi drug vzrok in to je funkcijska sočasnost.

Spoznali smo tudi podatkovno odvisne grafe (DDG), ki ponazarjajo odvisnost posameznih opravil med seboj. Primer takšnega grafa prikazuje slika 7.1: Vzemimo, da imamo zaporedno kodo, ki ustreza DDG na sliki:

```
double v, w, x, y;

v = alpha ();
```



Slika 7.1: Graf DDG, ki prikazuje odvisnost opravil.

```
w = beta ();
x = gamma (v, w);
y = delta ();
z = epsilon (x, y);
printf ("%6.2f\n", z);
```

Očitno so funkcije `alpha`, `beta` in `delta` neodvisne, zato se lahko izvajajo paralelno. S tem smo izkoristili vso funkcijsko paralelnost, ki nam jo ponuja graf. To lahko opišemo s pomočjo ukaza

```
#pragma omp parallel sections
```

Ta ukaz mora biti pred kodo, katere bloki se lahko izvajajo paralelno. Vsak tak blok pa mora biti označen z ukazom `section`. Primer:

```
#pragma omp parallel sections
{
    #pragma omp section
    v = alpha ();

    #pragma omp section
    w = beta ();

    #pragma omp section
    y = delta ();
}
x = gamma (v, w);
z = epsilon (x, y);
printf("%6.2f\n", z);
```

Za obravnavani DDG na sliki 7.1 lahko uporabimo funkcijski paralelizem tudi drugače. Najprej paralelno izvedemo samo opravili `alpha` in `beta`, nato pa po njunem zaključku še paralelno `gamma` in `delta`. Tedaj smo uporabili dve različni paralelni sekciji, ki si sledita ena za drugo. Na ta način lahko zmanjšamo število preslikav združi/razdruži in s tem presežek, ki vpliva na čas izvajanja programa. Vse štiri prireditve lahko tako združimo v eno sekcijo, pred katero postavimo ukaz `parallel`, ukaza `sections` pa postavimo tako, da ločimo prvi in drugi par funkcij, ki se lahko izvaja paralelno. Ta način funkcijske sočasnosti je podan s kodo:

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        v = alpha ();

        #pragma omp section
        w = beta ();
    }
    #pragma omp sections
    {
        #pragma omp section
        x = gamma (v, w);
        #pragma omp section
        y = delta ();
    }
}
z = epsilon (x, y);
printf ("%6.2f\n", z);
```

V enem pogledu je ta sočasnost boljša od prve, ker ima dve paralelni sekciji, ki vsaka zahtevata dve niti. Prva rešitev je imela le eno paralelno sekcijo, ki je zahtevala tri niti. Če imamo na voljo le dva procesorja, potem je druga rešitev lahko hitrejša. Ali je res hitrejša, pa je odvisno od časov izvajanja posameznih funkcij v sekcijah.

Primer paralelnega programa za množenje matrike in vektorja, ki uporablja ukaze knjižnice OpenMP, je prikazan na sliki 7.2. Primerjava s podobnim programa na sliki 6.6, ki uporablja knjižnico MPI, kaže na enostavnost uporabe knjižnice OpenMP.

```
#include <stdlib.h>
#include <omp.h>

void main(int argc, char* argv[])
{
    int m, n;
    float *X, *Y;
    float **A, *Astorage;
    int i, j;

    m = 10000; /* st. vrstic v matriki A */
    n = 10000; /* st. stolpcev v matriki A */

    /* dodelitev pomnilnika */
    X = (float *) malloc (n * sizeof (float)); /* vektor X */
    Y = (float *) malloc (m * sizeof (float)); /* vektor Y */
    Astorage = (float *) malloc (m * n * sizeof (float));
    A = (float **) malloc (m * sizeof (float *)); /* mat. A */
    for(i = 0; i < m; i++)
        A[i] = &Astorage [i * n];

    /* produkta matrike in vektorja */
    #pragma omp parallel for private (j) firstprivate (n)
    for (i = 0; i < m; i++)
    {
        Y[i] = 0.0;
        for (j = 0; j < n; j++ )
            Y[i] = Y[i] + A[i][j] * X[j];
    }

    /* sprostitev pomnilnika */
    free(X);
    free(Y);
    free(A);
    free(Astorage);
}
```

Slika 7.2: Uporaba knjižnice OpenMP pri množenju matrike in vektorja.

7.7 Hibridno programiranje s knjižnicama MPI in OpenMP

Mnogo komercialnih računalnikov ima danes dve ali več jeder (procesorjev), kar pomeni, da obstaja znotraj posameznega računalnika arhitektura s skupnim po-

mnilnikom. Več takih računalnikov lahko tudi združimo v večračunalniški sistem. V tem primeru je mogoče med procesorji s skupnim pomnilnikom izkoristiti prednosti, ki jih ponuja knjižnica OpenMP, pri porazdeljevanju opravil med računalniki pa prednosti knjižnice MPI.

Če imamo torej na razpolago gručo več računalnikov, ki imajo večprocesorsko arhitekturo, lahko realiziramo paralelni program s pomočjo knjižnice MPI, vendar s tem nismo izkoristili potencialne paralelnosti, ki se ponuja z večjedrno zasnovo procesorjev z deljenim pomnilnikom. Zato lahko dodatno MPI program delimo na niti, ki z uporabo knjižnice OpenMP še bolje izkoristijo računalniški sistem.

V mnogih primerih lahko s hibridnim programom, ki hkrati uporablja knjižnici MPI in OpenMP, pohitrimo izvajanje na račun manjšega komunikacijskega presežka glede na program, ki izkorišča le knjižnico MPI. Vzemimo, da izvajamo program na gruči večprocesorskih računalnikov, kjer ima vsak večprocesorski sistem k jeder. Da izkoristimo vsa jedra, mora program MPI kreirati $m \times k$ procesov, ki so aktivni med koraki komuniciranja. Po drugi strani pa mora hibridni program kreirati le m procesov. V paralelnih odsekih programa se breme deli med k niti v vsakem večprocesorskem sistemu. Tudi v tem primeru je izkoriščen vsak procesor, vendar je med koraki komunikacije aktivnih le m procesov. To pomeni, da je komunikacijski presežek manjši pri hibridnem programu, kot pri programu, ki uporablja samo knjižnico MPI.

Drugi razlog za hitrejšo izvedbo hibridnega programa je lahko izpolnjen takrat, kadar določena potencialna paralelnost algoritma zahteva klice časovno zamudnih funkcij MPI. Tedaj je bolje, če izvedemo replikacijo potencialno paralelnega dela algoritma in ga izvedemo paralelno z nitmi v okviru večprocesorskih računalnikov, kjer pa tudi pride do določenega presežka. Vzemimo primer, ko je nujni zaporedni delež v algoritmu 5 %, čisti paralelni delež 90 %, ostalih 5 % pa je lahko izvedenih paralelno s časovno zamudnimi funkcijami MPI. Vzemimo, da imamo na razpolago osem dvojedrnih večprocesorskih sistemov, skupaj torej $8 \times 2 = 16$ procesorjev. Po Amdahlovem zakonu je tedaj maksimalna pohitritev enaka

$$\psi_{\text{MPI}} = \frac{1}{0{,}10 + 0{,}90/16} = 6{,}4 \quad . \tag{7.1}$$

Po drugi strani lahko algoritem izvedemo na osmih MPI procesih in dovolimo, da vsak večprocesorski sistem računa na dveh nitih. Za 90 % kode je s tem paralelnost zagotovljena, saj 16 niti izvede ta del kode 16 krat hitreje. 5 % kode, ki je porazdeljena po nitih, bosta dve jedri izvedli dvakrat hitreje. 5 % časa za nujni zaporedni del pa ostane enak kot v primeru zaporednega programa. Maksimalna pohitritev je

$$\psi_{\text{MPI+OpenMP}} = \frac{1}{0{,}05 + 0{,}05/2 + 0{,}90/16} = 7{,}4 \quad , \tag{7.2}$$

iz česar sledi, da je hibridni program za 19 % hitrejši od programa, ki uporablja samo knjižnice MPI.

Poglavje 8

Paralelno procesiranje

V prejšnjih poglavjih smo spoznali, kako je mogoče prirediti zaporedno programsko kodo za izvajanje na paralelnih računalniških arhitekturah z uporabo specializiranih knjižnic MPI in OpenMP. V tem poglavju pa bomo spoznali možnost, s katero je možno paralelno procesiranje problema oziroma algoritma brez spreminjanja programske kode. Takšen pristop k porazdeljenem procesiranju je enostavnejši, zahteva pa specifično računalniško okolje, imenovano tudi GRID, in ustrezno programsko opremo (platformo), ki omogoča preslikavo zaporedne kode, opremljene z napotki (*ang.* script) v tako okolje.

Ime GRID izhaja iz angleške besede za omrežje. Pred časom se je namreč tudi v računalništvu porodila ideja, da bi uporabljali računalniško procesno moč in pomnilniške vire na podoben način kot uporabljamo, na primer, električno in vodovodno omrežje - torej z enostavnim priklopom nanj. Po analogiji naj bi bilo takšno računalniško omrežje uporabnikom tudi ves čas na voljo. Računalniško okolje GRID zato pomeni nedefinirano omrežje brezštevilnih računalniških sistemov, ki so na razpolago uporabnikom, ne da bi ti podrobno poznali omrežje in njegove kapacitete. Za uporabo takega omrežja je zato potrebna posebna programska oprema (platforma), ki povezuje uporabnike z okoljem GRID in rešuje probleme zasedenosti in rezervacij procesorske moči in pomnilniških virov.

V nadaljevanju bomo spoznali značilnosti okolja GRID, paradigmo komunikacije P2P (*ang.* peer-to-peer), pregledali njune skupne značilnosti in razlike, nazadnje pa si bomo podrobneje ogledali platformo Condor, skupaj s primerom za pisanje datoteke z napotki, na osnovi katere platforma Condor porazdeli zahtevana opravila v okolje GRID .

8.1 Okolje GRID in porazdeljeno računanje

Osnovni cilj računanja v okolju GRID je omogočiti delitev virov med uporabniki in institucijami. Pri tem so viri lahko pomnilnik, pasovna širina, procesorske zmogljivosti in podatki. Običajno ti viri pripadajo organizacijam in institucijam po celem svetu. V primeru okolja GRID viri tvorijo porazdeljene, heterogene in dinamične navidezne organizacije. S tem, ko okolje GRID vire navzven predstavi z ustreznimi servisi, postane omrežna infrastruktura za interoperabilno storitveno dejavnost. Prva ideja takšnega računalniškega okolja je bila servisno usmerjena znanost.

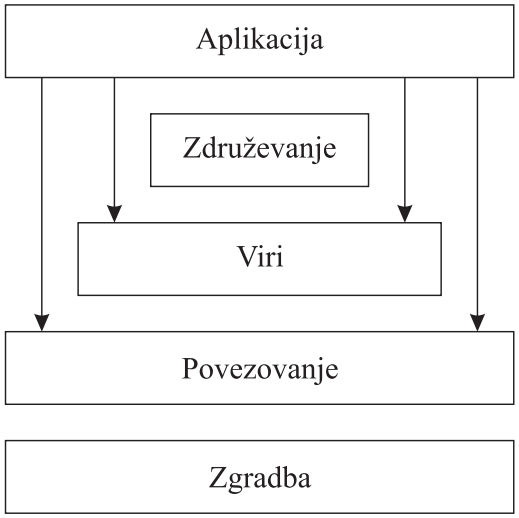
Okolje GRID je nastalo kot rezultat raziskav porazdeljenih sistemov. Razvoj Interneta in svetovnega spleta, skupaj s porazdeljenimi vmesnimi platformami (*ang.* middleware platforms), je omogočil obsežne porazdeljene aplikacije. Na tej točki sta se srečali disciplini, ki sta se ukvarjali z okoljem GRID in komunikacijami P2P. Primeri takšnih aplikacij so porazdeljeno superračunalništvo, na primer, simulacije fizikalnih procesov, visoko zmogljivo računanje zgolj z izkoriščanjem neuporabljenih procesnih ciklov namiznih računalnikov, računanje na zahtevo (za kratkotrajne zahteve in uravnoteženje procesnega bremena), podatkovno intenzivno računanje (izločanje informacij iz podatkov, ki so skladiščeni geografsko porazdeljeno) in kolaborativno računanje.

8.1.1 Arhitektura okolja GRID

Čeprav arhitektura okolja GRID ni standardizirana, pa se je s časom uveljavil splošni pogled na njene strukturne značilnosti. Tako je okolje GRID kot okolje za kompleksno računanje bolj formalno definirano kot strojna in programska infrastruktura, ki omogoča odvisno, konsistentno, dostopno in ceneno zagotavljanje računalniških zmogljivosti. Če različne institucije ali posamezniki sodelujejo v takšnem odnosu, potem tvorijo virtualno organizacijo. Okolje GRID torej omogoča sodelovanje preko institucionalnih meja. Tipična arhitektura okolja GRID je predstavljena na sliki 8.1 V posameznih plasteh se nahajajo različne komponente sistema glede na njihove funkcionalnosti in zmožnosti. Aplikacija v vrhnji plasti lahko na podlagi zahtev neposredno uporablja komponente iz plasti združevanja, virov in povezovanja.

Plast zgradbe (*ang.* Fabric Layer) omogoča dostopnost virov, ki jih posamezna vozlišča v okviru okolja GRID ponujajo v splošno uporabo. Viri se delijo na računske, pomnilne, omrežne, programske (*ang.* code storage) in na mape (*ang.* directories). Plast virov izvaja specifične operacije nad posameznimi viri in ponuja poenoten vmesnik za komunikacijo z višjimi plastmi.

Povezovalna plast gosti najpomembnejše varnostne in komunikacijske protokole.



Slika 8.1: Plasti v arhitekturi okolja GRID.

Omogoča podatkovno izmenjavo med viri v plasti zgradbe. Komunikacijski protokoli te plasti so običajno iz vrste TCP/IP protokolov. Varnostni protokoli v okolju GRID običajno izkoriščajo sistem javnih ključev.

Plast virov je odgovorna za operacije upravljanja virov (*ang.* resource management operations), kot so rezervacije, dostopi, vodenje, nadzor nad kvaliteto servisov oziroma storitev, zaračunavanje storitev. Dejanski viri, s katerimi ta plast upravlja, so pod nadzorom plasti zgradbe.

Združevalna plast je namenjena koordinaciji različnih skupin virov. Gosti komponente kot so na primer servisi map, razvrščanje in posredovanje storitev, nadzorovanje in diagnosticiranje storitev, servisiranje replikacije podatkov, vodenje obremenitev.

Končno aplikativna plast obsega uporabniške aplikacije, potrebne za izvajanje virtualne organizacije. Te aplikacije uporabljajo storitve, ki jih ponujajo nižje plasti, do katerih imajo neposreden dostop.

8.2 Sistemi P2P

Sistem enak z enakim ali sistem P2P (*ang.* peer-to-peer) je samo-organizirajoč sistem enakovrednih, avtonomnih entitet P (*ang.* peers), ki omogočajo deljeno uporabo porazdeljenih virov v omrežnem okolju ob izogibanju centralizirane kontrole. To je sistem s kompletno decentralizirano samo-organizacijo in uporabo virov.

Znotraj niza enakovrednih entitet (P) vsaka od njih uporablja vire, ki jih zagotavljajo ostale entitete, povezane v omrežje. Ob priključitvi entitete v omrežje se njen naslov določi dinamično. Sistemi P2P imajo torej svoj naslovni in poimenski prostor, ki se razlikuje od tradicionalnega Internetnega naslovnega prostora. Podatki v entitetah so opisani z nestrukturiranimi identifikatorji, ki izhajajo iz podatkov v razpršenih tabelah (*ang.* hash tables). To pomeni, da podatek ni naslovljen z lokacijo ampak z vsebino (*ang.* content-addressable storage). V primeru večkratnih kopij podatka lahko njegova poizvedba privede do katerekoli kopije. Sistemi P2P iščejo podatke z vsebino, za razliko od Interneta, kjer je povezovanje zasnovano na lokacijah entitet.

Uporaba deljenih virov zahteva od entitet interakcijo z vsemi ostalimi entitetami. V splošnem interakcije potekajo brez centralne kontrole, kar pomeni, da imajo sistemi P2P popolnoma decentralizirane krmilne mehanizme. Tukaj je entiteta hkrati odjemalec in strežnik. Entitete so enaki partnerji s simetrično funkcionalnostjo. Vsaka entiteta je glede na svoje vire popolnoma avtonomna.

Prva generacija sistemov P2P za izmenjavo datotek je bila predstavnica nestrukturiranih sistemov. To pomeni, da je imel centralni strežnik podatke o lokacijah posameznih datotek (*ang.* look-up table). Na osnovi podatka o lokaciji datoteke, ki ga je entiteti posredoval centralni strežnik, je sledil neposredni prenos datoteke med obema entitetama. Slaba stran teh sistemov je v velikem prometu proti strežniku in posledično v ozkem grlu glede virov, kot so pomnilnik, procesorska moč, pasovna širina. Poleg tega ti sistemi niso izpolnjevali pogoja skalabilnosti, saj ni bilo mogoče poljubno povečevati števila entitet.

Prav zahteva po skalabilnosti je privedla do razvoja nestrukturiranih sistemov P2P. Cilj pri razvoju nove generacije sistemov P2P je bil porazdeljen in decentraliziran samo-organizirajoč sistem, s pomnilnikom z vsebinskim naslavljanjem z uporabo razpršenih tabel.

8.3 Strukturni modeli omrežij

V procesu samo-organizacije sistemov P2P igrata pomembno vlogo omrežje SW (*ang.* Small World) in omrežje SF (*ang.* Scale Free), saj opisujeta strukturne lastnosti večine realnih omrežij. Omenili smo že, da osnovne funkcije sistemov P2P zajemajo komunikacijo, dodeljevanje virov in njihovo izmenjavo oziroma delitev. Za implementacijo teh funkcij mora sistem zagotavljati določene infrastrukturne pogoje:

- Decentralizacija: ne sme obstajati centralna točka, v kateri bi bile shranjene vse informacije o sistemu, podatkih in uporabnikih. Zato mora vsaka entiteta vsebovati vgrajena pravila, s katerimi se lahko priključi na sistem, prenaša

(*ang.* route) informacije ostalim entitetam ali zahteva poizvedbe po virih oziroma podatkih.

- Struktura: za učinkovito delitev podatkov in virov je pomembno, da je sistem strukturiran tako, da izboljša iskanje in povezovanje.
- Zanesljivost navkljub dinamičnim spremembam: skoraj vsi sistemi P2P so podvrženi konstantnim spremembam, priključevanju novih entitet oziroma odklapljanju ali začasnemu deaktiviranju obstoječih entitet. To zahteva mehanizme za stabilizacijo pomembnih lastnosti, kot sta na primer premer in povezljivost sistema.
- Skalabilnost: namen decentralizacije je, da je število vozlišč v sistemu lahko poljubno veliko. Sistem mora biti sposoben servisirati potrebe vseh vozlišč učinkovito in hitro, ne glede na njihovo število.

Ti štirje pogoji so posebno zanimivi, kadar gre za tako imenovana socialna omrežja, saj zanje velja tako samo-organizacija, kot tudi strukturna organizacija. Med realnimi omrežji sta danes zanimivi tudi omrežji Internet in svetovni splet. Raziskave teh omrežij so privedle do dveh pomembnih lastnosti razvijajočih in decentraliziranih omrežij, pa tudi do odgovora, kaj omogoča njihovo stabilnost in strukturiranost. Ti dve novi lastnosti, ki pomagata zagotavljati zgornje štiri pogoje, sta lastnost majhnega sveta in porazdeljenost. Omrežja, pri katerih je opazna lastnost majhnega sveta, bomo imenovali omrežja SW, omrežja, pri katerih opazimo porazdeljenost, pa bomo imenovali omrežja SF.

Sistemi P2P, ki ustvarijo omrežja s temi lastnostmi, praviloma ustrezajo tudi zgornjim pogojem. V nadaljevanju si bomo v kratkem ogledali značilnosti omrežij SW in SF, začeli pa bomo s splošnimi naključno povezanimi omrežji (*ang.* random networks).

Naključna omrežja

Pri naključnih omrežjih ali omrežjih RN (*ang.* random networks) je število vozlišč n fiksno. Poljubni dve vozlišči sta povezani z verjetnostjo p. V povprečju ima omrežje RN naslednje število povezav

$$e = p \begin{pmatrix} n \\ 2 \end{pmatrix} = p\frac{n(n-1)}{2} \quad . \tag{8.1}$$

Porazdelitev števila povezav na vozlišče k je binomska:

$$P(k) = \begin{pmatrix} n-1 \\ k \end{pmatrix} p^k (1-p)^{n-1-k} \quad , \tag{8.2}$$

kar pomeni, da je povprečna stopnja (*ang.* average degree) enaka:

$$\bar{k} = p(n-1) \approx pn \tag{8.3}$$

Za velike vrednosti n porazdelitev $P(k)$ postane podobna Poissonovi porazdelitvi. Ocena za povprečno najkrajšo pot $L(p)$, podana kot število vozlišč na poti med izbranima vozliščema, ki je globalna lastnost, izhaja iz izraza

$$\begin{aligned} \bar{k}^L &= n \\ L(p) &= \frac{\ln n}{\ln \bar{k}} \approx \frac{\ln n}{\ln(pn)} \quad . \end{aligned} \tag{8.4}$$

Povprečna najkrajša pot je v primeru omrežja RN majhna, kar je tipično tudi za omrežja SW. Koeficient grupiranja (*ang.* clustering coefficient) je enak

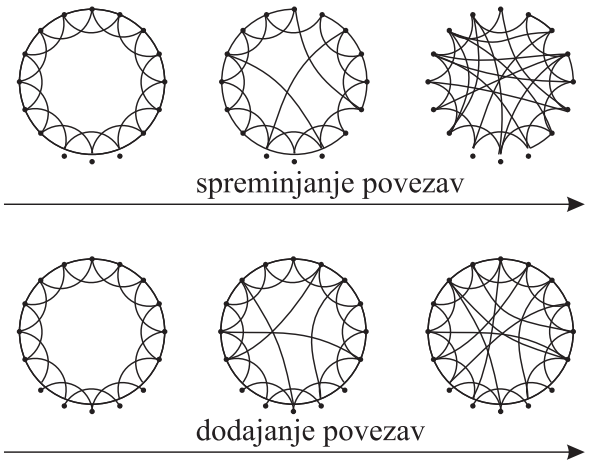
$$C(p) = p \approx \frac{\bar{k}}{n} \quad , \tag{8.5}$$

saj so povezave (*ang.* edges) v omrežju RN porazdeljene naključno. Koeficient grupiranja je pri istem številu vozlišč in povezav precej manjši kot pri omrežjih SW. Predstavlja lokalno lastnost, ki predstavlja povprečni delež povezav do sosednjih vozlišč.

Omrežja SW

Pri tej skupini omrežij je povprečna najkrajša pot med vozlišči tudi majhna, koeficient grupiranja pa precej večji kot pri omrežjih RN. Do omrežja SW lahko pridemo iz regularno povezanega omrežja, če z verjetnostjo $p \approx 0.1 - 0.3$ spremenimo regularne povezave v naključne. Prehod prikazuje slika 8.2.

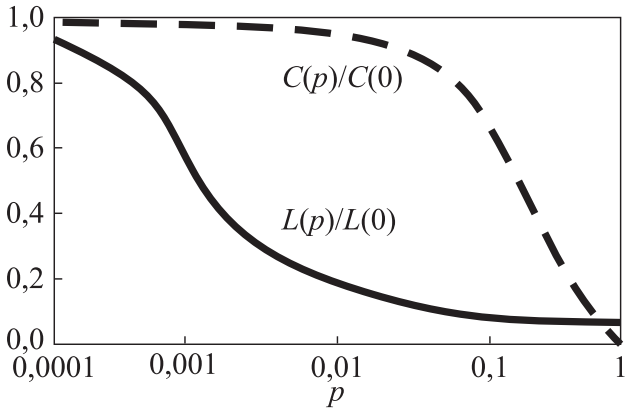
Oblika porazdelitve $P(k)$ je pri omrežjih SW podobna kot pri omrežjih RN, z vrhom pri $\bar{k}$. Veliko realnih omrežij ima lastnosti omrežij SW, kar je razvidno iz tabele 8.1, v kateri so navedeni koeficienti grupiranja C za različna omrežja, skupaj s parametri primerljivega omrežja RN. Tipične vrednosti ocene povprečne najkrajše poti L in koeficienta grupiranja C, relativno glede na izhodišče $p = 0$ za omrežje SW, prikazuje slika 8.3.



Slika 8.2: Prehod od regularnega omrežja v omrežje RN preko omrežja SW.

Tabela 8.1: Parametri realnih omrežij. Povzeto po [22].

omrežje	n	$\bar{k}$	Koeficient grupiranja C	
			izmerjeni	omrežje RN
Internet (avtonomni sistemi)	6374	3,8	0,24	0,00060
svetovni splet (strežniki)	153127	35,2	0,11	0,00023
električno omrežje	4941	2,7	0,080	0,00054
sodelovanje bioloških sistemov	1520251	15,5	0,081	0,000010
sodelovanje matematikov	253339	3,9	0,15	0,000015
sodelovanje filmskih igralcev	449913	113,4	0,20	0,00025
sočasnost izgovorjave besed	460902	70,1	0,44	0,00015
nevronske mreže	282	14,0	0,28	0,049
prehrambena veriga	134	8,7	0,22	0,065



Slika 8.3: Odvisnost ocene povprečne najkrajše poti L in koeficienta grupiranja C od verjetnosti p za omrežja SW. Povzeto po [23, 24].

Omrežje SF

Za večino obsežnih omrežij je porazdelitev povezav (*ang.* degree distribution) precej različna od Poissonove porazdelitve. Na primer, Internet in svetovni splet imata tako imenovano potenčno porazdelitev (*ang.* power law distribution)

$$P(k) \approx k^{-\gamma} \quad , \tag{8.6}$$

kjer je γ ustrezna konstanta. Do omrežij SF pridemo, če v naključnem omrežju število vozlišč narašča tako, da se vsako novo vozlišče poveže s starim vozliščem na osnovi verjetnosti, ki je proporcionalna številu povezav obstoječih vozlišč. To pomeni, da se nova vozlišča povezujejo z večjo verjetnostjo s tistimi, ki imajo večje število povezav. Temu načinu pravimo prednostni dostop (*ang.* preferential attachment). Vsako novo vozlišče i se želi povezati z obstoječimi vozlišči v omrežju, med katerimi je m_0 povezav. Verjetnost $\Pi(j)$, da obstoječe vozlišče j dobi eno od novih povezav, je enaka

$$\Pi(j) = \frac{k_t(j)n}{2m_t} \quad , \tag{8.7}$$

kjer je m_t število povezav v omrežju v času t, $k_t(j)$ pa trenutno število povezav vozlišča j.

8.4 Analiza okolij GRID in sistemov P2P

Po Fosterju predstavlja računalniško okolje GRID sistem, ki:

- koordinira vire, ki niso predmet centraliziranega nadzora,
- uporablja standard, odprte in splošne protokole ter vmesnike in
- ponuja netrivialno kvaliteto servisnih uslug.

Posebno prvi dve lastnosti sta značilni tudi na mnoge sisteme P2P. Celo na sistemskem nivoju razlika med okoljem GRID in sistemom P2P ni povsem jasna.

Primerjava okolij GRID in sistemov P2P na konceptualnem nivoju je še težja, ker predvsem ne obstaja univerzalno sprejeta definicija sistemov P2P. Poleg tega se ideja o računanju v okolju GRID neprestano razvija dalje. Pogosto se okolja GRID opisujejo s pomočjo mehanizmov, znanih iz sistemov P2P.

Sistem P2P je definiran kot samo-organizirajoč sistem enakih in avtonomnih entitet, ki po možnosti delujejo brez centralnega nadzora v komunikacijskem omrežju, z namenom delitve virov. Pri tem je poudarek na presečni uporabi virov. Po drugi definiciji je sistem P2P razred aplikacij, ki koristijo vire v okviru omrežja. Pojavljajo se tudi platforme sistemov P2P, ki zagotavljajo vmesno plast, neodvisno od operacijskega sistema in omogočajo posebno P2P deljeno uporabo virov.

8.4.1 Primerjava okolij GRID in sistemov P2P

Podobnosti in razlike

Osnovna motivacija za izdelavo aplikacij v okoljih GRID in sistemih P2P je podobna. Obe vrsti aplikacij se ukvarjata z združevanjem in organizacijo porazdeljenih virov, ki se delijo med uporabniki, povezanimi preko vsenavzočih (*ang.* ubiquitous) omrežij, kot je na primer Internet. Viri in storitve, ki so na voljo, so lahko locirani kjerkoli v sistemu in so transparentno dostopni uporabnikom na zahtevo.

Vendar obstajajo bistvene razlike na aplikacijskem, funkcionalnem in strukturnem nivoju. Aplikacije v okoljih GRID so v glavnem znanstvenega značaja in se uporabljajo v profesionalnem kontekstu. Število entitet je še vedno zmerno, sodelujoče institucije pa so običajno znane. Nasprotno pa aplikacije v sistemih P2P omogočajo odprt dostop do številnih in neznanih udeležencev s spremenljivim obnašanjem. Zato se tukaj pojavlja problem skalabilnosti in napak veliko bolj izrazito kot pri aplikacijah v okoljih GRID. Še vedno so v okviru sistemov P2P najbolj pogoste aplikacije, pri katerih gre za deljeno uporabo datotek in informacij ter za dostop do procesorske moči. Okolje GRID po drugi strani omogoča dostop do bazena virov, kot so na primer gruča računalnikov, pomnilni sistemi, podatkovne baze, znanstveni instrumenti in senzorji.

Okolja GRID in sistemi P2P: koncepta, ki konvergirata?

Danes je očitno, da koncept okolij GRID presega znanstvene aplikacije in da se kmalu lahko pričakuje njegova uporaba, na primer, v upravi, zdravstvu, administraciji, industriji, e-trgovanju. Z večjo uporabo okolja GRID se bo povečala tudi potreba pa skalabilnosti, odvisnosti in mehanizmih zaupanja, neobčutljivosti na napake samo-organizacije, samo-konfiguracije, samo-ozdravljanja. To pomeni, da bodo lahko mehanizmi, znani iz aplikacij v sistemih P2P, uporabljeni še bolj svobodno v okoljih GRID. Dejansko razvoj med sistemi P2P in storitvami svetovnega spleta ter med sistemi P2P in okolji GRID poteka paralelno.

Aplikacije, namenjene za sisteme P2P, se razvijajo v vedno kompleksnejših sistemih,

ki zagotavljajo vedno bolj zahtevne servisne usluge. Pojavljajo se nove in nove platforme, ki omogočajo generično podporo vedno naprednejšim aplikacijam za sisteme P2P. Takšna infrastruktura sistemov P2P ima veliko skupnega z infrastrukturo okolij GRID. Vendar pa cilj razvoja okolja GRID, to je ponuditi dostop do virov v obliki servisov, ni nujno povezan s temi platformami, ki so grajene predvsem za boljšo in fleksibilnejšo podporo aplikacijam.

Mehanizmi, značilni za sisteme P2P, lahko kmalu postanejo centralni mehanizmi v okoljih GRID. Aplikacije, namenjene sistemom P2P, bodo morale privzeti razvoj na osnovi platform za zadostno fleksibilnost v dinamičnem okolju. Čas bo pokazal, ali to pomeni konvergenco obeh področij in kakšen bo njun medsebojni vpliv v prihodnje.

8.5 Porazdeljeno procesiranje s sistemom Condor

Sistem Condor je programska oprema, ki omogoča visoko zmogljivo računanje z učinkovito izrabo delovnih postaj in namiznih računalnikov, medsebojno povezanih v računalniško omrežje. Sistem Condor lahko upravlja tako računalniške sisteme posebej namenjene zmogljivemu računanju, na primer gruče računalnikov, kot tudi splošno namenske računalniške sisteme, na primer, namenjene pisarniškemu delu. Razvit je bil na Univerzi Wisconsin-Madison in je prosto dostopen na spletni strani (http://www.cs.wisc.edu/condor/).

V primeru, da so v sistem Condor prijavljeni tudi namizni računalniki, je pomembno, da Condor uporabnika računalnika ne ovira pri običajnem delu. To pomeni, da sistem Condor lahko začne izkoriščati proste vire šele potem, ko le-ti določen čas niso v uporabi, takoj, ko je uporaba vira spet nakazana, pa mora sistem Condor svoja opravila zaključiti in celoten vir prepustiti uporabniku.

Ena najpomembnejših lastnosti sistema Condor je, da nam za izkoriščanje virov, ki jih ponuja, ni potrebno v ničemer spreminjati obstoječe sekvenčne programske kode. Ponovno prevajanje kode je potrebno samo v primerih, ko želimo uporabljati napredne funkcije sistema Condor. Za izvajanje v sistemu Condor so najbolj primerne konzolne aplikacije, programi z grafičnimi vmesniki predstavljajo le odvečno breme.

Viri, ki jih ponuja sistem Condor, so združeni v bazen (*ang.* pool). V istem bazenu so lahko združene zelo različne arhitekture: HP PA-RISC, Sun SPARC, SG MIPS, Intel x86, ALPHA, PowerPC, Intel Itanium, na katerih tečejo različni operacijski sistemi, med njimi: HP UX, Sun Solaris, SG IRIX, Red Hat Linux, Debian Linux, SuSE Linux, Windows XP, Windows 2000, Windows 2003, Tru64, MacOS, AIX.

Vsa komunikacija v sistemu Condor poteka preko upravnika bazena (*ang.* pool manager). Uporabnik lahko predloži (*ang.* submit) posel v sistem Condor iz kateregakoli računalnika v bazenu. Ta posel se preko upravnika prenese do virov,

kjer se izvaja. Ob zaključku se izhodni podatki preko upravnika prenesejo nazaj do računalnika iz katerega so bili posli predloženi. Neposredna komunikacija med viri ni mogoča. Komunikacija med programi poteka preko datotek. Prenašanje izvršilnih in vhodnih datotek do računskih virov in izhodnih datotek nazaj do računalnika, ki je posel predložil, lahko poteka z uporabo Condorjevih lastnih mehanizmov za prenos datotek, ali pa preko vzpostavljenega omrežnega datotečnega sistema (*ang.* network file system).

Potem, ko uporabnik predloži posel sistemu Condor, le-ta poišče proste računske vire, na katerih nato izvaja posle. Sistem Condor zna ugotoviti, da določen vir ni več na voljo. V tem primeru se izvajanje posla na tem viru zaustavi in prenese na drug vir, na katerem se izvajanje nadaljuje tam, kjer se je na prejšnjem viru končalo (*ang.* checkpointing). Sistem Condor ne zahteva prijavljanja na vire, na katerih izvaja posle. Dovolj je prijava v sistem Condor na računalniku, s katerega razpošiljamo posle, za vse nadaljnje aktivnosti pa poskrbi sistem sam z uporabo tehnologije oddaljenega klicanja procedur (*ang.* Remote System Call).

Za upravljanje z viri ima sistem Condor zelo dobro razvit mehanizem za povezovanje uporabnikov in ponudnikov virov. Vsi viri v sistemu Condor oglašujejo svoje lastnosti, tako statične kot dinamične, kot so velikost pomnilnika, tip in hitrost procesorja, operacijski sistem, trenutna obremenitev. Ob predložitvi posla v sistem uporabnik navede zahtevane lastnosti virov. Sistem Condor nato v funkciji posrednika poišče vire, ki ustrezajo zahtevam, in jih razvrsti glede na postavljene prioritete.

Moč sistema Condor se pokaže v primeru, ko moramo posel zagnati večkrat (več stokrat), po možnosti celo z različnimi začetnimi parametri. Vse posle namreč predložimo sistemu Condor z enim samim ukazom. Te posle lahko prevzamejo vsi prosti viri, ki ustrezajo podanim zahtevam, in jih paralelno izvajajo.

Pri kompleksnih obdelavah so posli pogosto medsebojno odvisni. Na primer, izbrani posel se lahko začne izvajati šele potem, ko so se zaključili potrebni posli pred njim. Razvrščanje poslov se opravi na podlagi odvisnosti med posli. Slednje se vpišejo v obliki usmerjenega necikličnega grafa (*ang.* directed acyclic graph, DAG), v katerem vsako vozlišče predstavlja posel, usmerjene povezave pa nakazujejo odvisnosti med posli.

Za boljšo podporo specializiranim sistemom Condor ponuja več različnih načinov delovanja, imenovanih vesolja (*ang.* Universe): Vanilla je najbolj osnovno, Standard ponuja nekaj več možnosti, recimo nadaljevanje izvajanja posla na drugem viru (*ang.* checkpointing), MPI ponuja večjo integracijo s knjižnico za paralelno programiranje MPI (*ang.* Message Passing Interface), PVM podobno ponuja boljšo integracijo s knjižnico za paralelno programiranje PVM (*ang.* Parallel Virtual Machines), tu je še Java za lažje delo s programi, napisani v jeziku Java, in Globus za povezavo sistema Condor s sistemom Globus GRID.

8.5.1 Pomembni ukazi sistema Condor

Za najpomembnejša opravila v sistemu Condor uporabljamo naslednje ukaze:

- `condor_store_cred add` prijavi uporabnika v sistem.
- `condor_status` prikaže osnovne podatke o trenutnem stanju virov v bazenu.
- `condor_q` izpiše vrsto, v kateri so navedeni posli, ki se že izvajajo, in tisti, ki še čakajo na izvedbo.
- `condor_submit datoteka.sub` sistemu Condor predloži posel, opisan v predložitveni datoteki `datoteka.sub`
- `condor_submit_dag datoteka.dag` predloži v sistem Condor posle z navedenimi odvisnostmi v obliki datoteke `datoteka.dag`.

Vsak od navedenih ukazov ima množico dodatnih argumentov, s katerimi vplivamo na način delovanja. Navedeni so v literaturi.

8.5.2 Primer uporabe sistema Condor

Poglejmo si numerično računanje integrala $\int_0^1 \frac{4}{1+x^2} dx = \pi$. Predpostavimo, da imamo na voljo imamo pet računalnikov. Računanje integrala lahko pohitrimo tako, da najprej na vsakem od petih računalnikov integral izračunamo na enem od intervalov, na koncu pa delne vsote seštejemo.

Za računanje delnega integrala imamo na voljo program `pi.exe`, ki ga iz ukazne vrstice zaženemo kot

```
pi.exe quad <index> <procNo> <accuracy>  .
```

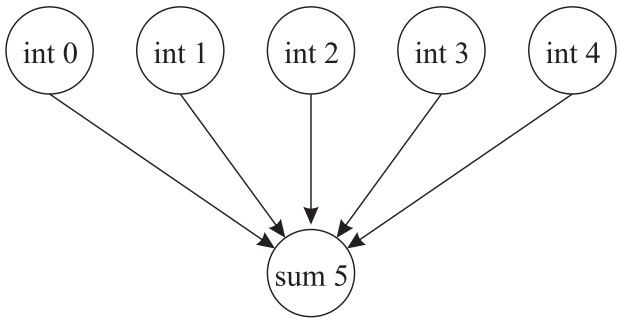
Argument `quad` določa, da gre za računanje delnega integrala, argument `<procNo>` določa število vseh območij, argument `<index>` pa določa območje integracije. V primeru petih območij velja: 0: [0,0, 0,2), 1: [0,2, 0,4), 2: [0,4, 0,6), 3: [0,6, 0,8) in 4: [0,8, 1,0). Z `<accuracy>` določimo natančnost aproksimacije. Program vrne integral na izbranem območju v datoteki `output_<index>.txt`. Program `pi.exe` lahko uporabimo tudi za seštevanje delnih integralov. V tem primeru ga kličemo kot

```
pi.exe sum <procNo>
```

kjer z argumentom `sum` zahtevamo seštevanje `<procNo>` delnih vsot, ki so zapisane v datotekah `output_0.txt` do `output_<procNo-1>.txt` in jih moramo sešteti. Program nam rezultat seštevanja vrne v datoteki `output_PI.txt`. Omenjeni izračun bi v ukazni vrstici lahko sekvenčno izvedli z naslednjimi klici programa `pi.exe`:

```
pi.exe int 0 5 10000000
pi.exe int 1 5 10000000
pi.exe int 2 5 10000000
pi.exe int 3 5 10000000
pi.exe int 4 5 10000000

pi.exe sum 5
```

Kadar imamo na voljo več virov, lahko integracije izvedemo paralelno, seštevanje pa šele potem, ko so integracije zaključene. Odvisnost med procesi lahko predstavimo z usmerjenim neciklič nim grafom na sliki 8.4, na katerem so procesi označeni s prvima dvema argumentoma programa pi.exe.



Slika 8.4: Usmerjeni neciklični graf za računanje števila π.

Sistemu Condor predložimo zahteve za obdelave v obliki predložitvenih datotek (*ang.* submit files). Predložitvena datoteka `int.sub`, ki sproži izračun delnih integracij na petih virih, izgleda takole:

```
Universe   = Vanilla
requirements = (opsys == "WINNT51") && (Memory > 250)
Executable = pi.exe
Arguments  = int $(Process) 5 10000000
Log        = std.log.txt
Output     = std.out.$(Process).txt
Error      = std.err.$(Process).txt
Queue 5
```

Spremenljivka `Universe` določa okolje (vesolje), v katerem se mora program izvajati. S spremenljivko `Requirements` podamo zahteve, ki jim morajo viri zadostiti, da se program na njih lahko izvaja. V našem primeru so zahtevani viri, na katerih teče operacijski sistem Windows XP (WINNT51) in imajo več kot 250 MB pomnilnika RAM. S spremenljivko `Executable` podamo program, ki se izvaja, v spremenljivko `Arguments` pa vpišemo vse njegove argumente. Nadalje s spremenljivkami `Log`, `Output` in `Error` podamo datoteko, v katero sistem Condor zapisuje podatke o izvajanju programa, datoteko, v katero sistem Condor ujame izpise na standardni izhod (zaslon, stdout), in datoteko, v kateri se zbirajo izpisi napak (stderr). Nazadnje z ukazom `Queue N` program predložimo v izvajanje na `N` virih. Pri delnih integracijah moramo za vsak vir pripraviti klic programa s svojimi argumenti. Sistem Condor v ta namen ponuja spremenljivko `$(Process)`, ki vsakemu viru priredi številčno oznako od 0 do N-1. Predložitvena datoteka `sum.sub` za seštevanje delnih vsot je zelo podobna:

```
Universe     = Vanilla
requirements = (opsys == "WINNT51") && (Memory > 250)
Executable = pi.exe
Arguments  = sum 5
Transfer_input_files = output_0.txt, output_1.txt, output_2.txt,\
                       output_3.txt, output_4.txt
Log  = std.log.txt
Output  = std.out.PI.txt
Error  = std.err.PI.txt
Queue 1
```

Za razliko od predložitvene datoteke `int.sub` v njej nastopa tudi spremenljivka `Transfer_input_files`, v katero vpišemo vse datoteke, ki jih mora sistem Condor poleg izvršilne datoteke prenesti na izbrani vir, da lahko izvede obdelavo.

Pri numerični integraciji moramo najprej izračunati delne vsote, šele nato jih lahko seštejemo. To odvisnost v sistem Condor vnesemo v obliki usmerjenega necikličnega grafa. Graf vpišemo kot tekstovno datoteko `pi.dag` z vsemi vozlišči in odvisnostmi med njimi:

```
Job I int.sub
Job S sum.sub
Parent I Child S
```

Z ukazom Job vsak posel predstavimo z vozliščem, ki mu podamo enostavno ime in pripadajočo predložitveno datoteko. Z ukazom `Parent-Child` podamo odvisnosti med posli. Posli, navedeni za besedo `Child`, se lahko izvedejo šele potem, ko so zaključeni vsi posli, ki so navedeni za besedo `Parent`.

Izračun lahko zaženemo na dva načina. Ena možnost je, da najprej izračunamo delne integrale. Izračun sprožimo s klicem

```
condor_submit int.sub
```

potem, ko se zaključi, pa s klicem

```
condor_submit sum.sub
```

sprožimo še seštevanje delnih vsot. Veliko bolj elegantno je zagnati celoten izračun s klicem

```
condor_submit_dag pi.dag    .
```

Poglavje 9

Trendi razvoja

S paralelnim programiranjem, na primer z uporabo knižnic MPI in OpenMP, in paralelnim procesiranjem, na primer v sistemu Condor, se zgodba o porazdeljenih sistemih seveda ne konča. Pravzaprav se na nek način šele prav začenja. Razvoj je namreč tudi na tem področju nezadržen in zato lahko v naslednjih letih pričakujemo precejšnje novosti in izboljšave. Da je temu res tako, bomo pokazali s predstavitvijo nekaterih trendov s tega področja. Predstavili bomo povsem nova področja mobilnega računanja (*ang.* mobile computing), ki izkoriščajo vse večjo povezljivost prenosljivih naprav, opremljenih s procesnimi elementi, vse-navzočega računanja (*ang.* ubiquitous computing), kot posledica povečane integracije računalniških naprav v vedno več produktov današnjega časa [25, 26, 27], ter možnosti, ki jih ponujajo najnovejši grafični procesorji [27].

9.1 Mobilno računanje

Mobilno računanje je paradigma o prenašanju osebnih računalnikov in ohranjanju povezljivosti z drugimi računalniki. Ta je postala izvedljiva v osemdesetih letih prejšnjega stoletja, ko je teža tedanjih računalnikov postala tako majhna, da je bil njihov prenos možen, po drugi strani pa je bilo mogoče oddaljene računalnike povezati preko telefonskih linij in modemov. Danes imajo prenosni računalniki številne vmesnike za brezžično povezovanje: infrardeči, WiFi, Bluetooth, GPRS, UMTS. Druga veja tehnološke evolucije pa je privedla do dlančnikov (*ang.* hand held computing), ki se prilegajo velikosti dlani in vključujejo osebni organizator PDA (*ang.* personal digital assistant), mobilne telefone in še druge specializirane ročno upravljane naprave, kot so kamera, radio, GPS. Dlančniki so v bistvu splošni računalniki, manjši od prenosnih računalnikov in omejeni s procesno močjo, velikostjo zaslona in lastnostmi drugih virov, glede prenosov pa imajo podobne brezžične komunicijske

zmožnosti kot prenosni računalniki.

Principi oziroma protokoli za brezžično komunikacijo morajo reševati dva bistvena problema:

1. kako ohraniti povezavo, če zaradi mobilnosti naprava prehaja iz ali v območje matičnih postaj (*ang.* base stations) in
2. kako omogočiti izbiro naprav za brezžično komunikacijo, če ne obstaja ustrezna infrastruktura (*ang.* ad hoc omrežja).

Če direktna brezžična povezava ni možna, je potrebna indirektna komunikacija preko drugih žičnih ali brezžičnih omrežnih segmentov. Pri tem sta pomembna dva faktorja: prvi je območje (*ang.* range) brezžičnega omrežja (čim večje je, tem več naprav tekmuje za svojo omejeno pasovno širino) in drugi energija, potrebna za prenos brezžičnega signala (proporcionalno je enaka kvadratu velikosti območja). Tehnologija sistemov P2P kot samo-organizirajoč sistem komuniciranja je pri tem uporabna tako za žična kot brezžična mobilna omrežja.

Vzemimo za ilustracijo mobilnega računanja vsakodnevni primer, to je klicanje taksija na ulici s pomočjo mobilnega telefona. Njegov uporabnik kliče centralo taksi agencije in naroči vozilo na trenutno lokacijo. Agencija pregleda lokacije svojih vozil in nato usmeri najbližje vozilo k stranki. To je primer kontekstno odvisne povezovalne sheme, ki v bistvu ne potrebuje centraliziranega elementa (agencije), zato je njegova implementacija mogoča tudi z P2P komunikacijo med mobilnimi računalniki (na primer GSM).

9.2 Vse-navzoče računanje

Izraz vse-navzoče računanje (*ang.* ubiquitous computing) je definiral Weiser leta 1991 [27], ko je, zaradi povečane prisotnosti računalnikov povsod okoli nas v tem videl revolucionarni premik v načinu uporabe računalnikov.

Kot prva v njegovem takratnem pogledu na bodočo uporabo računalnikov je bila napoved, da bo vsaka oseba lahko uporabljala mnogo računalnikov (ena oseba, mnogo računalnikov), kar ni danes nič nenavadnega. Slednje je mogoče primerjati s predhodno revolucijo osebnega računalništva (en računalnik za eno osebo). Vendar pri vsenavzočem računanju ne gre le za večje število računalnikov na osebo, ampak tudi za mnogo različnih računalnikov, tako po obliki kot funkciji, ki služijo različnim nalogam. Na primer, predpostavimo, da številne pripomočke za prikaz in pisanje (na primer, tabla, knjiga, papir, beležka, svinčnik), zamenjamo z različnimi računalniškimi zasloni in računalniškimi pisalniki. Tedaj bi se funkcije novih naprav

precej razširile, saj bi nam zaradi računalniške podpore omogočile še mnogo več kot osnovni rekviziti, na primer, e-zaslon bi pomagal pri risanju, e-knjiga bi omogočala poleg branja še iskanje pomenov besed, prevajanje, e-pisalo pa bi si zapomnilo napisan tekst in ga prenašalo drugemu uporabniku. Odpira se neskončno možnosti, toliko kot jih dopušča domišljija.

Naslednja Weiserjeva napoved pa je bila, da bodo računalniki izginili, oziroma da bodo zaradi vtkanosti v vse pore življenja postali nerazločljivi. To je bilo seveda mišljeno zgolj kot psihološki efekt nezadržnega razvoja računalniške tehnologije, vendar dobro odseva idejo, po kateri v prihodnje ne bomo razmišljali o računalniških zmogljivostih, na primer pralnega stroja ali avtomobila, pa čeprav ju nadzirajo številni vgrajeni mikroprocesorji. V določenih primerih je takšna neopaznost primerna, na primer v avtomobilu, pri drugih, predvsem pri mobilnih napravah, pa ne, saj je očitno, da prenosne telefone podpira računalnik, od katerega pričakujemo marsikatero pomoč ali funkcijo.

9.3 Grafični procesorji

Razvoj grafičnih procesorjev (*ang.* graphical processor unit, GPU) v zadnjih letih je bil izredno intenziven. Številna podjetja, predvsem Nvidia, AMD in Intel, so razvijala vedno bolj zmogljive grafične procesorje. Nekatera podjetja (Nvidia) so se specializirala za izdelovanje grafičnih kartic, druga pa so svoje grafične procesorje integrirala v okviru PC procesorjev. Zaradi intenzivnih raziskav v obeh smereh in velike konkurence so 2006 pri podjetju Nvidia vzporedno razvili programsko platformo CUDA (*ang.* Compute Unified Device Architecture) za splošno in masivno paralelno računanje (*ang.* massively parallel high-performance computing) na njihovih visoko zmogljivih grafičnih procesorjih [28]. Te grafične procesorje so zaradi nove aplikacije pri Nvidii imenovali tudi nitni procesorji (*ang.* thread processors).

Platforma CUDA vsebuje programska razvojna orodja za programski jezik C/C++, knjižnice funkcij, mehanizem za abstrakcijo strojne opreme, ki skriva strukturo grafičnih procesorjev pred programskimi razvijalci. Čeprav platforma CUDA zahteva od programerjev pisanje posebne kode za paralelno procesiranje, pa ne zahteva, da se eksplicitno naslavljajo posamezne niti, kar znatno poenostavi programiranje. Razvojna orodja za platformo CUDA delujejo skupaj s konvencionalnimi prevajalniki C/C++ tako, da lahko programer prepleta kodo za grafične procesorje s splošno kodo za centralni procesor. Koda za grafične procesorje na osnovi platforme CUDA je predvsem primerna za podatkovno intenzivne aplikacije (podatkovni paralelizem), ki jim zadoščajo matematične operacije v plavajoči vejici enojne natančnosti. Za novo generacijo grafičnih procesorjev pa je že predvidena dvojna natančnost operacij v plavajoči vejici.

Abstrakcija strojne opreme poenostavlja programiranje z zakrivanjem detajlov o arhitekturi grafičnih procesorjev. S tem si proizvajalec zagotovi popolno prenosljivost programov tudi na nove rodove grafičnih procesorjev s hipotetično popolnoma različno arhitekturo na strojnem nivoju. Za ilustracijo, grafični procesor podjetja Nvidia GeForce 8 ima 128 nitnih procesorjev, od katerih vsak lahko nadzira do 96 sočasnih niti, kar pomeni skupaj 12,288 niti.

Programski model knjižnice CUDA se znatno razlikuje od programske kode, namenjene za izvajanje na enonitnem centralnem procesorju. Pri eno-nitnih kodah procesor dostavlja en ukazni niz, ki se izvaja zaporedno nad podatki. Splošni model grafičnega procesorja (*ang.* general purpose GPU, GPGPU) omogoča uporabo točkovnih senčil (*ang.* shaders) oziroma splošnih aritmetičnih enot za računanje v enojni natančnosti. Procesiranje po modelu GPGPU izkazuje visoko stopnjo paralelnosti, vendar le-ta zavisi od pomnilnika izven grafičnega procesorja, s katerim mora slednji komunicirati, kar zmanjšuje hitrost procesiranja. Model CUDA deluje tako, da se podatki razdelijo na manjše bloke, ki se nahajajo v pomnilniku na čipu (*ang.* on-chip memory). To omogoča več-nitnim procesorjem, da si delijo vsak blok. Skupine niti se aktivirajo ciklično, glede na dosegljivost podatkov.

Pomembna lastnost knjižnic CUDA je tudi ta, da programerji ne pišejo eksplicitne nitne kode. Strojni nadzornik niti (*ang.* hardware thread manager) upravlja niti avtomatsko. Ukazi iz številnih niti si lahko delijo nitni procesorski cevovod v istem času in procesorji lahko preklapljajo pozornost med temi nitmi v enem ciklu ure. Upravljanje niti je za programerja transparentno. Poleg tega smrtni objem ni možen niti teoretično.

Prevajanje kode, ki vključuje knjižnice CUDA, ni enoznačno, saj zahteva več korakov, delno zaradi dveh ciljnih arhitektur (grafični procesor in procesor gostitelj) in delno zaradi abstrakcije na strojem nivoju. Program je običajno mešanica kode, napisane za centralni procesor in za grafične procesorje.

Ob koncu povejmo še, da je podjetje Nvidia do sedaj prodalo že 50 milijonov grafičnih procesorjev, ki podpirajo platformo CUDA. Na tržišču obstajajo tudi izdelki drugih proizvajalcev paralelnih platform za grafične procesorje, na primer PeakStream, RApidMind. S tem je niša za nove razvojne dosežke in visoko zmogljive paralelne aplikacije vedno bolj odprta.

Dodatek A

Koda programov MPI

A.1 Eratostenovo sito

```
/* Eratostenovo sito */

#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

/* najmanjsi indeks elementa, ki ga obdeluje proces id */
#define BLOCK_LOW(id,p,n)((id)*(n)/(p))

/* najvecji indeks elementa, ki ga obdeluje proces id */
#define BLOCK_HIGH(id,p,n)(BLOCK_LOW((id)+1,p,n)-1)

/* stevilo elementov, ki gjih obdeluje proces id */
#define BLOCK_SIZE(id,p,n)(BLOCK_LOW((id)+1,p,n)-BLOCK_LOW(id,p,n))


/*   Glavna funkcija */
int main (int argc, char *argv[])
{
    int    n;            /* Do katerega stevila iscemo prastevila */
    int    global_count; /* stevilo vseh prastevil */
    int    count;        /* lokalno stevilo prastevil */
    int    first;  /* lokalni indeks prvega veckratnika */
    int    low_value;    /* najmanjse stevilo pri procesu id */
```

```
int    high_value;   /* najvecje stevilo pri procesu id */
int    size;         /* stevilo elementov pri procesu id */
int    i;
int    index;        /* Index trenutno opazovanega stevila */
int    prime;        /* Trenutno prastevilo */
char   *marked;      /* del oznak za stevila */
int    p;            /* stevilo procesov */
int    id;           /* rang procesa */

MPI_Init (&argc, &argv);

MPI_Comm_rank (MPI_COMM_WORLD, &id);
MPI_Comm_size (MPI_COMM_WORLD, &p);
MPI_Barrier(MPI_COMM_WORLD);

/* Preko ukazne vrstice vnesemo do katerega
   stevila iscemo prastevila, eratosten <n> */
if (argc != 2) {
    if (id == 0) printf ("Sintaksa: %s <m>\n", argv[0]);
    MPI_Finalize();
    exit (1);
}
n = atoi(argv[1]);

/* vsak proces si izracuna obmocje, na katerem isce prastevila */
low_value = 2 + BLOCK_LOW(id,p,(n-1));
high_value = 2 + BLOCK_HIGH (id,p,(n-1));
size = BLOCK_SIZE (id, p, (n-1));

/* preverimo, ce proces 0 vkljucuje vsa stevila za sejanje */
if ((2 + (n-1)/p) < (int) sqrt((double) n)) {
    if (id == 0) printf ("Prevec procesov\n");
    MPI_Finalize();
    exit (1);
}

/* rezervacija pomnilnika za oznake */
marked = (char *) malloc (size);
for(i=0; i<size; i++)
    marked[i]=0;

/* inicializacija iskanja */
if (id == 0) index = 0;
    prime = 2;
```

```
    do {
        /* najprej moramo poiskati lokalni indeks, ki ustreza
           stevilu, pri katerem moramo zaceti oznacevati */
        if (prime * prime > low_value)
            first = prime * prime - low_value;
        else {
            if (!(low_value % prime)) first = 0;
            else first = prime - (low_value % prime);
        }

        /* oznacujemo veckratnike trenutnega prastevila po
           celotnem polju */
        for (i = first; i < size; i += prime) marked[i] = 1;

        /* proces 0 poisce prvo neoznaceno stevilo in
           ga poslje ostalim procesom*/
        if (id == 0) {
            while (marked[++index]);
            prime = index + 2;
        }
        if (p > 1) MPI_Bcast (&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
    } while (prime * prime <= n);

    /* Ko je oznacevanja konec, vsak process presteje
       svoja prastevila */
    count = 0;
    for (i = 0; i < size; i++)
        if (!marked[i]) count++;

    /* Proces 0 zbere skupno vsoto prastevil in jo izpise */
    MPI_Reduce (&count, &global_count, 1, MPI_INT,
                MPI_SUM, 0, MPI_COMM_WORLD);
    if (id == 0)
        printf ("Do stevila %d obstaja %d prastevil.\n",
                n, global_count);

    MPI_Finalize ();

    return 0;
}
```

A.2 Floydov algoritem

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

/* uporabni makroji */
#define BLOCK_LOW(id,p,n)((id)*(n)/(p))
#define BLOCK_SIZE(id,p,n)(BLOCK_LOW((id)+1,p,n)-BLOCK_LOW(id,p,n))
#define BLOCK_OWNER(j,p,n)(((p)*((j)+1)-1)/(n))
#define MIN(x, y)(((x) < (y))? (x): (y))

/* funkcija prebere matriko iz datoteke in jo razposlje          */
/* ostalim procesom                                              */
/* za branje datoteke name je zadolzen proces p-1                */
/* funkcija vrne parametre B, Bstorage, m in n (zato dodatna *)  */
/* primer zapisa matrike 6 x 6v datoteki:                        */
/*      6                                                        */
/*      6                                                        */
/*      0 2 5 1000 1000 1000                                     */
/*      1000 0 7 1 1000 8                                        */
/*      1000 1000 0 4 1000 1000                                  */
/*      1000 1000 1000 0 3 1000                                  */
/*      1000 1000 2 1000 0 3                                     */
/*      1000 5 1000 2 4 0                                        */
/*  velika stevilka (1000), oznacuje, da vozlisci nista povezani */
void readAndDistributeMatrix(char *name, float ***B,
                             float **Bstorage,
                             int *m, int *n)
{
    int i, j;
    FILE *file;             /* kazalec na datoteko */
    int p;                  /* stevilo procesov */
    int id;                 /* rang procesa */
    int localRows;          /* stevilo vrstic na lokalnem procesu */
    MPI_Status status;      /* rezultat MPI_Recv */

    MPI_Comm_size (MPI_COMM_WORLD, &p);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);

    if (id == p-1)
    {
        file = fopen (name, "r");
```

```
        if (file == NULL)
            *m = 0;
        else
        {
            fscanf( file, "%d", m);
            fscanf( file, "%d", n);
        }
    }

    MPI_Bcast (m, 1, MPI_INT, p-1, MPI_COMM_WORLD);
    if (*m == 0)
        MPI_Abort (MPI_COMM_WORLD, -1);
    MPI_Bcast (n, 1, MPI_INT, p-1, MPI_COMM_WORLD);

    localRows = BLOCK_SIZE(id, p, *m);
    *Bstorage = (float *) malloc (*n * localRows * sizeof(float));
    *B = (float **) malloc(localRows * sizeof(float *));
    for (i = 0; i < localRows; i++ )
        (*B)[i] = &(*Bstorage)[*n * i];

    if (id == p-1)
    {
        for (i = 0; i < p-1; i++)
        {
            for (j = 0; j < *n * BLOCK_SIZE (i, p, *m); j++)
                fscanf(file, "%f", &(*Bstorage)[j]);
            MPI_Send (*Bstorage, *n * BLOCK_SIZE (i, p, *m),
                      MPI_FLOAT, i, 0, MPI_COMM_WORLD);
        }
        for (j = 0; j < *n * localRows; j++)
            fscanf(file, "%f", &(*Bstorage)[j]);
        fclose (file);
    }
    else
        MPI_Recv (*Bstorage, *n * localRows, MPI_FLOAT,
                  p-1, 0, MPI_COMM_WORLD, &status);
}

/* izpis matrike B velikosti m x n na zaslon */
void printMatrix( float **B, int m, int n)
{
    int i, j;

    for (i = 0; i < m; i++)
```

```
    {
        for(j = 0; j < n; j++)
            printf ("%8.3f ", B[i][j]);
        printf ("\n");
    }
}

/* Proces p-1 sprejme vse dele matrike od ostalih procesov in */
/* jih izpise. Parametri funkcije so lokalna matrika in       */
/* dimenziji celotne matrike                                  */
void printDistributedMatrix( float **B, int m, int n)
{
    int i, j;
    float **C;          /* deli matrike od ostalih procesov */
    float *Cstorage;    /* pomnilniski prostor za matriko */
    int localRows;      /* stevilo lokalnih vrstic */
    int maxBlockSize;   /* najvecje stevilo vrstic v bloku */
    int p;              /* stevilo procesov */
    int id;             /* rang procesa */
    MPI_Status status;  /* status za MPI_Recv */

    MPI_Comm_size (MPI_COMM_WORLD, &p);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);

    localRows = BLOCK_SIZE (id, p, m);
    if (id == p-1)
    {
        if (p > 1)
        {
            maxBlockSize = BLOCK_SIZE (p-1, p, m);
            Cstorage = (float *) malloc
                       (n*maxBlockSize*sizeof(float));
            C = (float **) malloc (maxBlockSize * sizeof(float *));
            for (i = 0; i < maxBlockSize; i++)
                C[i] = &Cstorage[i * n];
            for (i = 0; i < p-1; i++)
            {
                MPI_Recv(Cstorage, n * BLOCK_SIZE (i, p, m),
                         MPI_FLOAT, i, 0, MPI_COMM_WORLD, &status);
                printMatrix(C, BLOCK_SIZE (i, p, m), n);
            }
            free( C );
            free( Cstorage );
        }
```

```
        printMatrix(B, localRows, n);
        printf ("\n");
    }
    else
        MPI_Send(*B, n * localRows, MPI_FLOAT,
                 p-1, 0, MPI_COMM_WORLD);
}

/* izracun najkrajsih poti po Floydovem algoritmu */
void computeShortestPaths (int id, int p, float **A, int n)
{
    int root;       /* proces, ki oddaja vrstico */
    int offset;     /* lokalni indeks odajane vrstice */
    float *Ak;      /* shranjuje oddajano vrstico */
    int i, j, k;

    Ak = (float *) malloc (n * sizeof(float));
    for (k = 0; k < n; k++)
    {
        root = BLOCK_OWNER (k, p, n);
        if (root == id )
        {
            offset = k - BLOCK_LOW (id, p, n);
            for (j = 0; j < n; j++)
                Ak[j] = A[offset][j];
        }
        MPI_Bcast (Ak, n, MPI_FLOAT, root, MPI_COMM_WORLD);
        for (i = 0; i < BLOCK_SIZE (id, p, n); i++)
            for (j = 0; j < n; j++)
                A[i][j] = MIN (A[i][j], A[i][k] + Ak[j]);
    }
    free (Ak);
}

/* glavni program */
int main (int argc, char *argv[])
{
    float *Astorage;    /* lokalni elementi sosednostne matrike */
    float **A;          /* kazalci na vrstice v sosed. matriki */
    int m, n;           /* stevilo stolpcev in vrstic v matriki */
    int id;             /* rang procesa */
    int p;              /* stevilo procesov */
    int i, j, k;
```

```
    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &p);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);

    readAndDistributeMatrix("matrix.txt", &A, &Astorage,
                            &m, &n); /* branje matrike */

    if( m != n )
    {
        MPI_Finalize();
        exit(1);
    }
    printDistributedMatrix(A, m, n);    /* izpis */
    computeShortestPaths(id, p, A, n);  /* izracun najkrajsih poti */
    printDistributedMatrix(A, m, n);    /* izpis */

    free(A);                            /* sprostitev pomnilnika */
    free(Astorage);
    MPI_Finalize();
}
```

Dodatek B

Funkcije knjižnice MPI

V tem dodatku so opisane vse funkcije ki jih ponuja knjižnica MPI po standardu 1.1. Parametri funkcij so označeni kot

- `IN`, če gre za vhodne parametre, ki jih mora priskrbeti uporabnik.
- `OUT`, če gre za izhodne parametre. Uporabnik mora pripraviti ustrezno strukturo v katero funkcija vpiše vrednosti.
- `IN/OUT`, če lahko uporabnik in funkcija spreminjata vrednost.

Komentarji spremenljivk so zaradi omejitev urejevalnika besedil napisani brez šumnikov.

```
int MPI_Abort (
    MPI_Comm comm,  /* IN - komunikator */
    int error_code  /* IN - koda napake */
)
```

Funkcija poskuša prekiniti procese v podanem komunikatorju. Okolju MPI vrne izbrano kodo napake.

```
int MPI_Address (
    void *location,   /* IN - lokacija v 'offsets' */
    MPI_Aint *offsets /* IN - polje naslovov */
)
```

Funkcija vrne naslov v bajtih za `location` v polju naslovov. Namenjena je gradnji sestavljenih podatkovnih tipov.

```
int MPI_Allgather (
    void *send_buffer,       /* IN - naslov podatkov za posiljanje */
    int send_cnt,            /* IN - stevilo elementov */
    MPI_Datatype send_dtype, /* IN - podatkovni tip posiljanih
                                     elementov */
    void *recv_buffer,       /* OUT - naslov sprejetih podatkov */
    int recv_cnt,            /* IN - stevilo sprejetih elementov */
    MPI_Datatype recv_dtype, /* IN - podatkovni tip sprejetih
                                     elementov */
    MPI_Comm comm            /* IN - komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Vsak proces, ki je vpleten v komunikacijo, združi `send_cnt` elementov od vseh ostalih procesov. Ko se funkcija konča, so elementi zbrani v `recv_buffer`. V primeru, da je število elementov od procesa do procesa različno, je potrebno uporabiti funkcijo `MPI_Allgatherv`.

```
int MPI_Allgatherv (
    void *send_buffer,       /* IN - naslov podatkov za posiljanje */
    int send_cnt,            /* IN - stevilo elementov */
    MPI_Datatype send_dtype, /* IN - podatkovni tip posiljanih
                                     elementov */
    void *recv_buffer,       /* OUT - naslov sprejetih podatkov */
    int *recv_cnts,          /* IN - vektor s stevilom sprejetih
                                     elementov od vsakega procesa*/
    int *recv_disp,          /* IN - vektor odmikov od zacetka
                                     recv_buffer kamor se shranjujejo
                                     sprejeti elementi od vsakega
                                     procesa */
    MPI_Datatype recv_dtype, /* IN - podatkovni tip sprejetih
                                     elementov */
    MPI_Comm comm            /* IN - komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Število elementov, ki jih združimo v skupno strukturo `recv_buffer`, je od procesa do procesa lahko različno. Zato v polje `recv_disp` vpišemo, kam v strukturo naj se vpišejo podatki. Položaj je podan kot odmik od začetka strukture `recv_buffer`. V primeru, da je število elementov, ki jih prispevajo procesi, enako, je bolje uporabiti funkcijo `MPI_Allgather`.

```
int MPI_Allreduce (
    void *send_buffer,       /* IN - naslov podatkov za posiljanje */
    void *recv_buffer,       /* OUT - naslov sprejetih podatkov */
```

```
    int cnt,                  /* IN - stevilo elementov */
    MPI_Datatype send_dtype, /* IN - podatkovni tip posiljanih
                                    elementov */
    MPI_op op                 /* IN - operator redukcije */
    MPI_Comm comm             /* IN - komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Izvaja operacijo redukcije nad podatki. Ko se funkcija zaključi, imajo vsi procesi rezultat operacije redukcije. V primeru, da rezultat potrebuje en sam proces, je bolje uporabiti funkcijo `MPI_Reduce`.

```
int MPI_Alltoall (
    void *send_buffer,       /* IN - naslov podatkov za posiljanje */
    int send_cnt,            /* IN - stevilo elementov */
    MPI_Datatype send_dtype, /* IN - podatkovni tip posiljanih
                                    elementov */
    void *recv_buffer,       /* OUT - naslov sprejetih podatkov */
    int recv_cnt,            /* IN - stevilo sprejetih elementov */
    MPI_Datatype recv_dtype, /* IN - podatkovni tip sprejetih
                                    elementov */
    MPI_Comm comm            /* IN - komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Izvaja izmenjavo podatkov med vsemi procesi v komunikatorju. Vsak proces odda enako število elementov vsakemu procesu in od vsakega procesa sprejme enako število elementov. Če je število elementov med procesi različno, je potrebno uporabiti funkcijo `MPI_Alltoallv`.

```
int MPI_Alltoallv (
    void *send_buffer,       /* IN - naslov podatkov za posiljanje */
    int *send_cnts,          /* IN - vektor s stevilom elementov,
                                    ki jih odda vsak proces */
    int *send_disp,          /* IN - vektor odmijov za oddane
                                    elemente */
    MPI_Datatype send_dtype, /* IN - podatkovni tip posiljanih
                                    elementov */
    void *recv_buffer,       /* OUT - naslov sprejetih podatkov */
    int *recv_cnts,          /* IN - vektor s stevilom sprejetih
                                    elementov od vsakega procesa*/
    int *recv_disp,          /* IN - vektor odmikov od zacetka
                                    recv_buffer kamor se shranjujejo
                                    sprejeti elementi od vsakega
```

```
                                    procesa */
    MPI_Datatype recv_dtype, /* IN - podatkovni tip sprejetih
                                    elementov */
    MPI_Comm comm            /* IN - komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Izvaja izmenjavo podatkov med vsemi procesi v komunikatorju. Vsak proces lahko odda različno število elementov vsakemu procesu in od vsakega procesa sprejme različno število elementov. Če je število elementov med procesi enako, je bolje uporabiti funkcijo `MPI_Alltoall`.

```
int MPI_Attr_delete (
    MPI_Comm comm, /* IN - komunikator */
    int key        /* IN - oznaka atributa */
)
```

Funkcija izbriše atribut v pomnilniku, ki mu ustreza oznaka `key`.

```
int MPI_Attr_get (
    MPI_Comm comm, /* IN - komunikator */
    int key,       /* IN - oznaka atributa */
    void *attr,    /* OUT - kazalec na atribut */
    int *flag      /* OUT - zastavica, ki ozna\v cuje
                          obstoj atributa */
)
```

Funkcija preko strukture `attr` vrne kazalec na predhodno shranjen atribut z oznako `key`. V primeru, da je branje atributa uspešno, se zastavica `flag` postavi na 1, drugače pa je 0.

```
int MPI_Attr_put (
    MPI_Comm comm, /* IN - komunikator */
    int key,       /* IN - oznaka atributa */
    void *attr     /* OUT - kazalec na atribut */
)
```

Funkcija preko strukture `attr` zapiše atribut z oznako `key` v pomnilnik.

```
int MPI_Barrier (
    MPI_Comm comm /* IN - komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Za vse procese v komunikatorju izvede sinhronizacijo z zapornico.

```
int MPI_Bcast (
    void *buffer,       /* IN/OUT - naslov v pomnilniku */
    int cnt,            /* IN - stevilo elementov */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    int root,           /* IN - rank procesa, ki posilja */
    MPI_Comm comm       /* IN - komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Omogoča, da izbrani proces `root` vsem ostalim procesom v komunikatorju pošlje sporočilo.

```
int MPI_Bsend (
    void *buffer,       /* IN - naslov posiljane strukture */
    int cnt,            /* IN - stevilo elementov */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    int root,           /* IN - rang procesa, ki posilja */
    MPI_Comm comm       /* IN - komunikator */
)
```

Funkcija izvaja operacijo pošiljanja z uporabo vmesnega pomnilnika (*ang.* buffer). Če proces izvaja pošiljanje z uporabo vmesnega pomnilnika in ostali procesi sporočila še niso sprejeli, lahko proces, ki pošilja, nadaljuje izvajanje. V primeru, da vmesnega pomnilnika zmanjka, pride do napake. Količino vmesnega pomnilnika lahko nastavimo preko `MPI_Buffer_attach`.

```
int MPI_Bsend_init (
    void *buffer,       /* IN - naslov posiljane strukture */
    int cnt,            /* IN - stevilo elementov */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    int dest,           /* IN - rang ciljnega procesa */
    int tag,            /* IN - oznaka sporo\v cila */
    MPI_Comm comm,      /* IN - komunikator */
    MPI_Request *handle /* IN - kazalec na zahtevo */
)
```

Funkcija vzpostavi stalno zahtevo za komunikacijo za `MPI_Bsend`. Uporabna je, kadar program neprestano izvaja komunikacijo z enakimi argumenti. Pošiljanje sporočila se sproži s klicem `MPI_Start`.

```
int MPI_Buffer_attach (
    void *buffer, /* IN - kazalec na vmesni pomnilnik */
    int size      /* IN - stevilo bajtov v vmesnem
                          pomnilniku */
)
```

S to funkcijo povežemo vmesni pomnilnik z ustrezno strukturo v pomnilniku. Običajno vmesni pomnilnik predstavlja strukturo, ki jo program dodeli s klicem funkcije `malloc`.

```
int MPI_Buffer_detach (
    void *buffer, /* OUT - kazalec na vmesni pomnilnik */
    int *size     /* OUT - stevilo bajtov v vmesnem
                           pomnilniku */
)
```

S klicem te funkcije MPI sprosti vmesni pomnilnik. Funkcija vrne kazalec na vmesni pomnilnik in njegovo velikost.

```
int MPI_Cancel (
    MPI_Request handle /* IN - kazalec na ustrezno sporocilo */
)
```

Funkcija označi za preklic nezaključeno operacijo pošiljanja ali sprejemanja. Funkcija se običajno zaključi takoj, sporočilo pa je priklicano kasneje.

```
int MPI_Cart_coords (
    MPI_Comm comm, /* IN - kartezicni komunikator */
    int rank,      /* IN - rang procesa */
    int dims,      /* IN - dimenzionalnost */
    int *coords    /* OUT - koordinate */
)
```

Funkcija vrne koordinate procesa v kartezični mreži.

```
int MPI_Cart_create (
    MPI_Comm old_comm, /* IN - komunikator */
    int dims,          /* IN - dimenzionalnost */
    int *size,         /* IN - polje velikosti dims s
                             stevilom procesov v vsaki
                             dimenziji */
    int *periodic,     /* IN - za vsako dimenzijo oznacimo
                             ali je periodicna */
    int reorder,       /* IN - dovoljenje za spremembo rangov */
    MPI_Comm cart_comm /* OUT - kazalec na kartezicni
                              komunikator */
)
```

Funkcija vrne kazalec na kartezični komunikator, v katerem so procesi urejeni v kartezično topologijo. Če je `reorder` nič, potem so rangi procesov enaki kot pri

starem komunikatorju. V nasprotnem primeru sistem spremeni range procesorjev in tako naredi preslikavo procesov na procesorje bolj učinkovito. Če predlagana kartezična topologija vključuje več procesov kot stari komunikator, funkcije vrne napako. Če je število procesov manjše kot pri starem komunikatorju, nekateri procesi dobijo oznako `MPI_COMM_NULL` in se zaključijo.

```
int MPI_Cart_get (
    MPI_Comm comm, /* IN - kartezicni komunikator */
    int dims,      /* IN - dimenzionalnost */
    int *size,     /* OUT - polje velikosti dims s
                            stevilom procesov v vsaki
                            dimenziji */
    int *periodic, /* OUT - za vsako dimenzijo podatek o
                            periodicnosti */
    int *coord     /* OUT - koordinate klicocega procesa */
)
```

Funkcija vrne podatke o kartezičnem komunikatorju in koordinate klicočega procesa v njem.

```
int MPI_Cart_map (
    MPI_Comm comm, /* IN - kartezicni komunikator */
    int dims,      /* IN - dimenzionalnost */
    int *size,     /* IN - polje velikosti dims s
                           stevilom procesov v vsaki
                           dimenziji */
    int *periodic, /* IN - za vsako dimenzijo podatek o
                           periodicnosti */
    int *new_rank  /* OUT - optimizirani rang */
)
```

Funkcija vrne optimirane range procesov za kartezični komunikator.

```
int MPI_Cart_rank (
    MPI_Comm comm, /* IN - kartezicni komunikator */
    int *coords,   /* IN - koordinate procesa */
    int *rank      /* OUT - rang procesa */
)
```

Funkcija vrne rang procesa s podanimi koordinatami v kartezičnem komunikatorju.

```
int MPI_Cart_shift (
    MPI_Comm comm, /* IN - kartezicni komunikator */
    int shift_dim, /* IN - dimenzija v kateri naredimo zamik *?
```

```
    int direction, /* IN - 0: gor, 1: dol */
    int *src,      /* OUT - vir sprejetega sporocila */
    int *dest      /* OUT - ponor oddanega sporocila */
)
```

Za procese v kartezičnem komunikatorju funkcija za izbrano dimenzijo pove range procesov, ki pošiljajo in ki sprejemajo sporočila. Če dimenzija ni periodična, funkcija vrne `MPI_PROC_NULL` v `src` ali `dest`.

```
int MPI_Cart_sub (
    MPI_Comm comm,     /* IN - kartezicni komunikator */
    int *free,         /* IN - polje podatkov o dimenzijah
                             i = 1 - koordinata i se lahko
                                     spremeni
                             i = 0 - koordinate i mora
                                     ostati nespremenjena */
    MPI_Comm *new_comm /* OUT - novi komunikator */
)
```

Funkcija razdeli kartezični komunikator na več mrež manjše dimenzije. Število dimenzij v novem komunikatorju je enako številu elementov `free[i]`, ki so enaki "0". Kazalec na novi komunikator predstavlja `new_comm`.

```
int MPI_Cartdim_get (
    MPI_Comm comm, /* IN - kartezi\v cni komunikator */
    int *dim       /* OUT - stevilo dimenzij */
)
```

Funkcija vrne število dimenzij v kartezičnem komunikatorju.

```
int MPI_Comm_compare (
    MPI_Comm comm1, /* IN - komunikator 1 */
    MPI_Comm comm2, /* IN - komunikator 2 */
    int *result     /* OUT - rezultat primerjave */
)
```

Funkcija primerja dva komunikatorja med seboj. Rezultat primerjave je lahko `MPI_IDENT`, če sta enaka, `MPI_CONGRUENT`, če sta konteksta različna, komunikatorja pa vsebujeta enako število procesov z enakimi rangi, `MPI_SIMILAR`, če so konteksti in rangi različni, število procesov pa je enako, ali `MPI_UNEQUAL` v vseh ostalih primerih.

```
int MPI_Comm_create (
    MPI_Comm old_comm, /* IN - stari komunikator */
    MPI_Group group,   /* IN - skupina procesov */
    MPI_Comm *new_comm /* OUT - novi komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Iz procesov, navedenih v `group`, naredi nov komunikator.

```
int MPI_Comm_dup (
    MPI_Comm old_comm, /* IN - stari komunikator */
    MPI_Comm *new_comm /* OUT - novi komunikator */
)
```

Funkcija vrne nov komunikator z enako skupino, vendar različnim kontekstom.

```
int MPI_Comm_free (
    MPI_Comm comm, /* IN - komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Sprosti vire, povezane s komunikatorjem `comm`.

```
int MPI_Comm_group (
    MPI_Comm comm,    /* IN - komunikator */
    MPI_Group *group  /* OUT - skupina procesov */
)
```

Funkcija vrne skupino procesov s komunikatorjem `comm`.

```
int MPI_Comm_rank (
    MPI_Comm comm, /* IN - komunikator */
    int *rank      /* OUT - rang klicocega procesa */
)
```

Funkcija vrne rang klicočega procesa.

```
int MPI_Comm_remote_group (
    MPI_Comm comm,   /* IN - inter-komunikator */
    MPI_Group *procs /* OUT - Kazalec na oddaljeno skupino */
)
```

Funkcija vrne oddaljeno skupino procesorjev za inter-komunikator `comm`. Glej `MPI_Intercomm_create`.

```
int MPI_Comm_remote_size (
    MPI_Comm comm, /* IN - inter-komunikator */
    int *size      /* OUT - velikost oddaljene skupine */
)
```

Funkcija vrne velikost oddaljene skupine, ki je povezana v inter-komunikator. Glej `MPI_Intercomm_create`.

```
int MPI_Comm_size (
    MPI_Comm comm, /* IN - komunikator */
    int *size      /* OUT - stevilo procesov v komunikatorju */
)
```

Funkcija vrne število procesov v komunikatorju.

```
int MPI_Comm_split (
    MPI_Comm old_comm, /* IN - stari komunikator */
    int partition,     /* IN - stevilka delitve */
    int new_rank,      /* IN - vrednost ranga */
    MPI_COMM *comm     /* OUT - nov komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Funkcija razdeli procese v aktivnem komunikatorju na več skupin. Procesi z isto vrednostjo `partition` padejo v isto skupino. Znotraj skupine so procesi rangirani glede na vrednost `new_rank`. Funkcija vsakemu procesu vrne novi komunikator, h kateremu je dodeljen.

```
int MPI_Comm_test_inter (
    MPI_Comm comm, /* IN - komunikator */
    int *flag      /* OUT - rezultat testa */
)
```

Funkcija vrne 1, če gre za inter-komunikator, in 0 drugače. Glej `MPI_Intercomm_create`.

```
int MPI_Dims_create (
    int nodes, /* IN - stevilo vozlisc v mrezi */
    int dims,  /* IN - stevilo dimenzij */
    int *size  /* OUT - velikost vsake dimenzije */
)
```

Funkcija vrne polje števil, ki podajajo število vozlišč v vsaki dimenziji mreže, tako da so dimenzije po velikosti kar se da enakovredne.

```
int MPI_Errhandler_create (
    MPI_Handler_function *eh_func, /* IN - kazalec na funkcijo
                                        za obdelovanje napak */
    MPI_Errhandler *eh             /* OUT - kazalec na objekt za
                                        delo z napakami */
)
```

Po klicu funkcije je `eh_func` funkcija, ki se izvaja ob prekinitvah. Funkcija vrne kazalec na napako. Uporabniške funkcije za obravnavanje napak morajo imeti glavo oblike `void MPI_Handler_function(MPI_Comm *, int *, ...)`. Prvi argument funkcije je vedno komunikator, drugi koda napake, ki naj bi jo vrnil MPI program, število in pomen ostalih argumentov pa sta odvisna od implementacije knjižnice MPI.

```
int MPI_Errhandler_free (
    MPI_Errhandler *eh /* IN - kazalec na objekt za
                              delo z napakami */
)
```

Funkcija sprosti pomnilnik za objekt za delo z napakami in nastavi `eh` na `MPI_ERRHANDELER_NULL`.

```
int MPI_Errhandler_get (
    MPI_Comm comm,          /* IN - komunikator */
    MPI_Errhandler *eh_func /* IN - kazalec na funkcijo za
                                   obdelovanje napak */
)
```

Funkcija za klicoči proces poveže funkcijo za obravnavanje napak s komunikatorjem.

```
int MPI_Errhandler_set (
    MPI_Comm comm,     /* IN - komunikator */
    MPI_Errhandler *eh /* IN - kazalec na objekt za delo
                              z napakami /*
)
```

Funkcija za klicoči proces poveže objekt za obravnavanje napak s komunikatorjem.

```
int MPI_Error_class (
    int code,  /* IN - koda napake */
    int *class /* OUT - razred napake */
)
```

Razredi napak so del standarda MPI. Funkcija vrne razred napake, ki ima kodo `code`.

```
int MPI_Error_string (
    int err_code,          /* IN - napaka */
    char *err_string,      /* OUT - niz znakov */
    int *err_string_length /* OUT - stevilo znakov v nizu */
)
```

Funkcija vrne opis napake `err_code`. Pred klicem funkcije mora program za niz znakov dodeliti pomnilnik velikosti vsaj `MPI_MAX_ERROR_STRING` bajtov.

```
int MPI_Finalize (void)
```

Funkcija zapre povezavo s sistemom MPI. Vsak proces mora pred zaključkom poklicati to funkcijo.

```
int MPI_Gather (
    void *send_buffer,        /* IN - naslov posiljane strukture */
    int send_cnt,             /* IN - stevilo elementov za
                                      posiljanje */
    MPI_Datatype send_dtype,  /* IN - podatkovni tip elementov za
                                      posiljanje */
    void *recv_buffer,        /* OUT - naslov strukture za sprejem */
    int recv_cnt,             /* IN - stevilo elementov za sprejem */
    MPI_Datatype recv_dtype,  /* IN - tip sprejetih podatkov */
    int root,                 /* IN - rang posiljatelja */
    MPI_Comm comm             /* IN - komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Funkcija izvaja operacijo zbiranja, pri kateri proces `root` združi `send_cnt` elementov od vsakega procesa, vključno s samim seboj. Funkcija vrne sestavljeno strukturo v `recv_buffer`. V primeru, da različni procesi vrnejo različno število elementov, je potrebno uporabiti funkcijo `MPI_Gatherv`.

```
int MPI_Gatherv (
    void *send_buffer,        /* IN - naslov posiljane strukture */
    int send_cnt,             /* IN - stevilo elementov za
                                      posiljanje */
    MPI_Datatype send_dtype,  /* IN - podatkovni tip elementov za
                                      posiljanje */
    void *recv_buffer,        /* OUT - naslov strukture za sprejem */
    int *recv_cnt,            /* IN - polje s stevilom podatkov za
                                      sprejem od vsakega procesa */
    int *recv_disp,           /* IN - polje polozajev na katere
                                      naj se v recv_buffer vpisejo
                                      sprejeti podatki */
    MPI_Datatype recv_dtype,  /* IN - tip sprejetih podatkov */
    int root,                 /* IN - rang posiljatelja */
    MPI_Comm comm             /* IN - komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Funkcija izvaja operacijo zbiranja, pri kateri proces `root` združi `send_cnt[j]` elementov od vsakega procesa `j`, vključno s samim seboj. Funkcija vrne sestavljeno strukturo v `recv_buffer`. V strukturo `recv_buffer` se elementi procesa `j` vpišejo na polje, ki je `recv_dsip[j]` odmaknjeno od začetka strukture. V primeru, da različni procesi vrnejo enako število elementov, je bolje uporabiti funkcijo `MPI_Gather`.

```
int MPI_Get_count (
    MPI_Status *status, /* IN - rezultat sprejemanja */
    MPI_Datatype dtype, /* IN - tip sprejetih elementov */
    int* cnt            /* OUT - stevilo sprejetih elementov */
)
```

Funkcijo pokličemo s parametrom `status`, ki ga vrne funkcija za sprejem, na primer `MPI_Recv`, in s tipom sprejetih elementov. Vrne število sprejetih elementov.

```
int MPI_Get_elements (
    MPI_Status *status, /* IN - rezultat sprejemanja */
    MPI_Datatype dtype, /* IN - tip sprejetih elementov */
    int* pe_cnt         /* OUT - stevilo sprejetih elementov */
)
```

Funkcijo pokličemo s parametrom `status`, ki ga vrne funkcija za sprejem, na primer `MPI_Recv`, in s tipom sprejetih elementov. Vrne število sprejetih primitivnih elementov.

```
int MPI_Get_processor_name (
    char *name, /* OUT - ime procesorja */
    int *length /* OUT - dolzina imena */
)
```

Funkcija vrne ime procesorja, na katerem se izvaja proces.

```
int MPI_Get_version (
    int *major, /* OUT - stevilka verzije (1 ali 2)*/
    int *minor  /* OUT - stevilka podverzije */
)
```

Funkcija vrne verzijo sistema MPI na računalniku.

```
int MPI_Graph_create (
    MPI_Comm comm_old,   /* IN - stari komunikator */
    int n,               /* IN - stevilo vozlisc v grafu */
    int *degree,         /* IN - polje velikosti n s
```

```
                                stopnjo vozlisca.
                                0 - stopnja vozlisca 0,
                                i - stopnja vozlisca i je
                                degree[i] - degree[i-1] */
    int *edge,           /* IN - ostali podatki o vozliscih */
    int reorder,         /* IN - 1: dovoljeno spreminjanje rangov */
    MPI_Comm *comm_graph /* OUT - novi komunikator */
)
```

Funkcija vrne kazalec na komunikator v obliki grafa. Če je reorder 0, se rangi procesov ne spremenijo. Vir vsake povezave med vozlišči lahko določimo iz polja `degree`. Ponor vozlišča `i` določa `edge[i]`.

```
int MPI_Graph_get (
    MPI_Comm comm,   /* IN - komunikator v obliki grafa */
    int n,           /* IN - stevilo vozlisc v grafu */
    int m,           /* IN - stevilo povezav med vozlisci */
    int *index,      /* OUT - informacija o indeksu */
    int *edge        /* OUT - informacija o povezavi */
)
```

Funkcija vrne topologijo komunikatorja v obliki grafa `comm`, ki ima `n` vozlišč in `m` povezav.

```
int MPI_Graph_map (
    MPI_Comm comm, /* IN - komunikator v obliki grafa */
    int n,         /* IN - stevilo vozlisc v grafu */
    int *index,    /* IN - informacija o indeksu */
    int *edge      /* IN - informacija o povezavi */
    int *new_rank  /* OUT - novi rang procesa */
)
```

Funkcija poskuša optimirati razporeditev procesov po procesorjih za podani komunikator v obliki grafa. Vrne novi rang klicočega procesa. Novi rang ima vrednost `MPI_UNDEFINED`, če klicoči proces ni član komunikatorja v obliki grafa.

```
int MPI_Graph_neighbors (
    MPI_Comm comm,     /* IN - komunikator v obliki grafa */
    int rank,          /* IN - rang procesa */
    int max_neighbors, /* IN - najvecje stevilo sosedov */
    int *neighbors     /* OUT - rangi sosedov */
)
```

Funkcija vrne procese, ki so v grafu sosedje procesa `rank`.

```
int MPI_Graph_neighbors_count (
    MPI_Comm comm, /* IN - komunikator v obliki grafa */
    int rank,      /* IN - rang procesa */
    int *neighbors /* OUT - stevilo sosedov */
)
```

Funkcija vrne število sosedov, ki jih ima vozlišče `rank`.

```
int MPI_Graphdims_get (
    MPI_Comm comm, /* IN - komunikator v obliki grafa */
    int *vertices, /* OUT - vozlisca grafa */
    int *edges     /* OUT - povezave v grafu */
)
```

Funkcija vrne število vozlišč in število usmerjenih povezav med njimi za komunikator v obliki grafa.

```
int MPI_Group_compare (
    MPI_Group group1, /* IN - prva skupina procesov  */
    MPI_Group group2, /* IN - druga skupina procesov  */
    int *result       /* OUT - rezultat primerjave */
)
```

Funkcija primerja dve skupini procesov in vrne `MPI_IDENT`, če imata skupini enake procese z istimi rangi, `MPI_SIMILAR`, če imata enake procese, vendar z različnimi rangi, ali `MPI_UNEQUAL` v ostalih primerih.

```
int MPI_Group_difference (
    MPI_Group group1,    /* IN - prva skupina procesov  */
    MPI_Group group2,    /* IN - druga skupina procesov  */
    MPI_Group group_diff /* OUT - skupina razlik */
)
```

Funkcija vrne novo skupino procesov, v kateri so samo procesi, ki so v prvi skupini in ne v drugi. Procesi so urejeni na enak način kot v prvi skupini.

```
int MPI_Group_excl (
    MPI_Group group,    /* IN - skupina procesov  */
    int excl_num,       /* IN - stevilo procesov, ki jih zelimo
                              izkljuciti iz skupine */
    int *excl_ranks,    /* IN - rangi procesov, ki jih zelimo
                              izkljuciti */
    MPI_Group group_new /* OUT - skupina brez izklucenih procesov */
)
```

Funkcija iz skupine procesov `group` izključi navedene procese. Vrne kazalec na skupino, iz katere so navedeni procesi izključeni.

```
int MPI_Group_free (
    MPI_Group *group, /* IN - skupina procesov  */
)
```

Funkcija označi skupino za sprostitev in spremeni kazalec na skupino v `MPI_GROUP_NULL`. Pomnilnik ne bo sproščen, dokler se ne zaključijo vsi procesi, ki skupino uporabljajo.

```
int MPI_Group_incl (
    MPI_Group *group, /* IN - skupina procesov  */
    int new_size,     /* IN - stevilo procesov v novi skupini */
    int *old_ranks,   /* IN - rangi procesov v novi skupini */
    MPI_Group *new    /* OUT - nova skupina procesov */
)
```

Funkcija iz obstoječe skupine procesov ustvari novo, ki je lahko manjša od obstoječe. Velikost skupine je podana z `new_size`. V novi skupini so samo procesi z rangi, navedenimi v polju `old_ranks`. V novi skupini ima prvi proces v polju rang 0, zadnji pa `new_size-1`.

```
int MPI_Group_intersection (
    MPI_Group *group1,   /* IN - prva skupina procesov  */
    MPI_Group *group2,   /* IN - druga skupina procesov  */
    MPI_Group *group_new /* OUT - nova skupina procesov */
)
```

Funkcija vrne novo skupino procesov, v kateri so samo procesi, ki so tako v skupini `group1`, kot tudi v skupini `group2`. Procesi so urejeni tako kot v prvi skupini.

```
int MPI_Group_range_excl(
    MPI_Group *old,   /* IN - obstojeca skupina procesov  */
    int n,            /* IN - stevilo obmocij */
    int range[][3],   /* IN - obmocje */
    MPI_Group *new    /* OUT - nova skupina procesov */
)
```

Funkcija iz skupine procesov `old` izključi vse procese, ki so znotraj navedenih območij. Procesi v novi skupini so urejeni tako kot v stari skupini. Območje je podano kot `[first, last, step]`. Na primer, območje `[1, 7, 2]` sestavljajo procesi 1, 3, 5 in 7.

```
int MPI_Group_range_incl(
    MPI_Group *old,   /* IN - obstojeca skupina procesov  */
    int n,            /* IN - stevilo obmocij */
    int range[][3],   /* IN - obmocje */
    MPI_Group *new    /* OUT - nova skupina procesov */
)
```

Funkcija iz skupine procesov `old` izlušči vse procese, ki so znotraj navedenih območij. Procesi v novi skupini so urejeni tako kot v stari skupini. Območje je podano kot `[first, last, step]`. Na primer, območje `[1, 7, 2]` sestavljajo procesi 1, 3, 5 in 7.

```
int MPI_Group_rank(
    MPI_Group *group, /* IN - skupina procesov  */
    int *rank         /* OUT - rang procesov */
)
```

Funkcija vrne rang klicočega procesa v skupini.

```
int MPI_Group_size(
    MPI_Group *group, /* IN - skupina procesov  */
    int *size         /* OUT - stevilo procesov */
)
```

Funkcija vrne število procesov v skupini.

```
int MPI_Group_translate_ranks(
    MPI_Group *group1, /* IN - prva skupina procesov  */
    int n,             /* IN - stevilo rangov za primerjavo */
    int *rank1,        /* IN - polje rangov */
    MPI_Group *group2, /* IN - druga skupina procesov */
    int *rank2         /* OUT - stevilo rangov v drugi skupini */
)
```

Funkcija za podane range procesov v prvi skupini vrne ustrezne številke rangov v drugi skupini.

```
int MPI_Group_union (
    MPI_Group *group1,   /* IN - prva skupina procesov  */
    MPI_Group *group2,   /* IN - druga skupina procesov  */
    MPI_Group *group_new /* OUT - nova skupina procesov */
)
```

Funkcija vrne novo skupino procesov, v kateri so vsi procesi iz prve in druge skupine. Najprej so navedeni vsi procesi iz prve skupine, nato pa še procesi iz druge skupine, ki jih ni v prvi skupini.

```
int MPI_Group_Ibsend (
    void *buffer,       /* IN - kazalec na vmesni pomnilnik */
    int cnt,            /* IN - stevilo elementov v sporocilu */
    MPI_Datatype dtype, /* IN - podatkovni tip elementov */
    int dest,           /* IN - rang ciljnega procesa */
    int tag,            /* IN - oznaka sporocila */
    MPI_Comm comm,      /* IN - komunikator */
    MPI_Request *handle /* OUT - kazalec na zahtevo */
)
```

Funkcija začne takojšnje pošiljanje z uporabo vmesnega pomnilnika. S kazalcem `handle` lahko opazujemo status funkcije. Ker se izvajanje programa nadaljuje takoj po klicu funkcije, vmesnega pomnilnika ne smemo spreminjati, dokler funkcija ne konča dela.

```
int MPI_Init (
    void *argc,  /* IN - prvi parameter funkcije main */
    char ***argv /* IN - drugi parameter funkcije main */
)
```

Funkcija vzpostavi paralelno okolje MPI. Vsak program, ki uporablja funkcije MPI, mora poklicati `MPI_Init` pred klicem katerekoli druge funkcije MPI. Vsak proces sme poklicati funkcijo `MPI_Init` samo enkrat. S parametroma `argc` in `argv` lahko preko funkcije main iz ukazne vrstice funkciji pošljemo parametre za inicializacijo.

```
int MPI_Init_thread (
    void *argc,   /* IN - prvi parameter funkcije main */
    char ***argv, /* IN - drugi parameter funkcije main */
    int desired,  /* IN - zelena stopnja podpore za niti */
    int *provided /* OUT - stopnja podpore za niti, ki je na voljo */
)
```

Funkcija inicializira okolje MPI na enak način kot funkcije `MPI_Init`. Če uporabljamo funkcijo `MPI_Init_thread`, potem ni potrebno klicati funkcije `MPI_Init`. Tretji parameter je želena stopnja podpore za niti, ki je lahko

- `MPI_THREAD_SINGLE` - izvajanje samo na eni niti,
- `MPI_THREAD_FUNNELED` - proces je lahko več niten, toda samo glavna nit izvaja klice funkcij MPI,
- `MPI_THREAD_SERIALIZED` - vse niti lahko kličejo funkcije MPI, vendar se klici izvajajo zaporedno,
- `MPI_THREAD_MULTIPLE` - vse niti lahko kličejo funkcije MPI, klici se izvajajo vzporedno.

Funkcija preko zadnjega parametra vrne stopnjo podpore, ki jo nudi sistem. Standard ne zahteva, da knjižnice MPI podpirajo delo z nitmi.

```
int MPI_Initialized (
    int *flag, /* OUT - podatki o inicializaciji  */
)
```

Funkcija vrne `flag=1`, če je proces inicializiran in `flag=0` drugače. To je edina funkcija, ki jo lahko kličemo pred funkcijo `MPI_Init`.

```
int MPI_Intercomm_create (
    MPI_Comm local_comm,  /* IN - lokalni komunikator */
    int local_leader,     /* IN - rang glavnega procesa v lokalnem
                                  komunikatorju */
    MPI_Comm remote_comm, /* IN - oddaljeni komunikator */
    int remote_leader,    /* IN - rang glavnega procesa v oddaljenem
                                  komunikatorju */
    int tag,              /* IN - oznaka */
    MPI_Comm *new_comm    /* OUT - inter-komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Funkcija iz lokalnega komunikatorja, katerega član je proces, in oddaljenega komunikatorja (proces ni njegov član) sestavi nov komunikator. Ta se imenuje inter-komunikator. Oznaka `tag` je potrebna za ločevanje klicev funkcije, kadar se hkrati ustvarja več inter-komunikatorjev. Komunikacija med procesi v obeh komunikatorjih je možna vsaj preko obeh glavnih procesov.

```
int MPI_Intercomm_merge (
    MPI_Comm inter,  /* IN - inter-komunikator */
    int high,        /* IN - oznaka skupine */
    MPI_Comm *intra  /* OUT - intra-komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Funkcija inter-komunikator spremeni v običajen komunikator. Vsi procesi v eni skupini inter-komunikatorja morajo postaviti `high=1`, vsi procesi v drugi pa `high=0`. V novem komunikatorju so najprej navedeni procesi s `high=0`. Glej `MPI_Intercomm_create`.

```
int MPI_Iprobe (
    int src,           /* IN - rang posiljatelja */
    int tag,           /* IN - oznaka sporocila */
    MPI_Comm comm,     /* IN - komunikator */
    int *flag,         /* OUT - zastavica uspesnosti prenosa */
    MPI_Status *status /* OUT - kazalec na status */
)
```

Funkcija preverja, če je na voljo novo sporočilo, vendar ga ne sprejme. Funkcija ne čaka, da je sporočilo na voljo. V primeru, da je sporočilo na voljo, postavi `flag=1`. Več o sporočilu lahko preberemo v strukturi `status`. Funkcija je uporabna v primeru, da želimo sporočilu dodeliti natančno toliko pomnilnika, kot ga potrebuje. Namesto točno določenega pošiljatelja lahko uporabimo `MPI_ANY_SOURCE`, namesto točno določene oznake pa `MPI_ANY_TAG`.

```
int MPI_Irecv (
    void *buffer,       /* OUT - naslov vmesnega pomninika za
                               sprejem */
    int cnt,            /* IN - stevilo elementov za sprejem */
    MPI_Datatype dtype, /* IN - podatkovni tip elementov */
    int src,            /* IN - posiljatelj sporocila */
    int tag,            /* IN - oznaka sporocila */
    MPI_Comm comm,      /* IN - komunikator */
    MPI_Request *handle /* OUT - kazalec na zahtevo */
)
```

Funkcija zahteva sprejem sporočila, če je le-ta na voljo, nato pa vrne nadzor sistemu. Do vmesnika `buffer` se ne sme dostopati, dokler sprejem ni zaključen. Za čakanje na zaključek lahko uporabimo funkcijo `MPI_Wait`.

```
int MPI_Irsend (
    void *buffer,       /* IN - naslov sporocila */
    int cnt,            /* IN - stevilo elementov v sporocilu */
    MPI_Datatype dtype, /* IN - podatkovni tip elementov */
    int dest,           /* IN - sprejemnik sporocila */
    int tag,            /* IN - oznaka sporocila */
    MPI_Comm comm,      /* IN - komunikator */
    MPI_Request *handle /* OUT - kazalec na zahtevo */
)
```

Funkcija sproži takojšnje pošiljanje sporočila. Vsebina `buffer` se ne sme spreminjati, dokler prenos ni zaključen. Preko kazalca `handle` lahko preverimo, kdaj je prenos končan.

```
int MPI_Isend (
    void *buffer,       /* IN - naslov sporocila */
    int cnt,            /* IN - stevilo elementov v sporocilu */
    MPI_Datatype dtype, /* IN - podatkovni tip elementov */
    int dest,           /* IN - sprejemnik sporocila */
    int tag,            /* IN - oznaka sporocila */
    MPI_Comm comm,      /* IN - komunikator */
    MPI_Request *handle /* OUT - kazalec na zahtevo */
)
```

Funkcija zahteva od sistema, da začne pošiljati sporočilo in zaključi delo. Vsebina `buffer` se ne sme spreminjati, dokler prenos ni zaključen. Preko kazalca `handle` lahko preverimo, kdaj je prenos končan. Izvajanje funkcije moramo zaključiti s klicem drugih funkcij, na primer funkcije `MPI_Wait`.

```
int MPI_Issend (
    void *buffer,       /* IN - naslov sporocila */
    int cnt,            /* IN - stevilo elementov v sporocilu */
    MPI_Datatype dtype, /* IN - podatkovni tip elementov */
    int dest,           /* IN - sprejemnik sporocila */
    int tag,            /* IN - oznaka sporocila */
    MPI_Comm comm,      /* IN - komunikator */
    MPI_Request *handle /* OUT - kazalec na zahtevo */
)
```

Funkcija vzpostavi takojšen sinhroni prenos - sistemu ne vrne nadzora, dokler se ne začne izvajati ustrezen sprejem. Dokler se pošiljanje ne zaključi, sporočila ne smemo spreminjati.

```
int MPI_Keyval_create (
    MPI_Copy_function *copy_fn,  /* IN - kazalec na funkcijo za
                                         kopiranje atributov */
    MPI_Delete_function *del_fn, /* IN - kazalec na funkcijo za
                                         brisanje atributov */
    int *key,                   /* OUT - kazalec na atribut */
    void *extra                 /* IN - dodatna informacija za
                                         povratne klice */
)
```

Funkcija ustvari nov atribut, označen s `key`.

```
int MPI_Keyval_free (
    int *keyval /* IN - vrednost kljuca */
)
```

Funkcija označi navedeno vrednost ključa za brisanje in postavi `keyval = MPI_KEYVAL_INVALID`.

```
int MPI_Op_create (
    MPI_User_function *func, /* IN - kazalec na asociativno
                                     funkcijo */
    int commutative,         /* IN - komutativnost funkcije */
    MPI_Op *op               /* OUT - kazalec na operator */
)
```

S pomočjo te funkcije lahko definiramo svoj globalen operator redukcije, ki mora biti asociativen. Če je operator komutativen, je potrebno nastaviti `commutative = 1`. Funkcija vrne kazalec na operand `op`. Funkcija, ki definira nov operator redukcije, mora imeti naslednjo glavo

```
void MPI_User_function(void *in_vector,
                       void *in_out_vector,
                       int *length,
                       MPI_Datatype *dtype)
```

Naj $u[0], \ldots, u[\texttt{length} - 1]$ predstavljajo elemente vektorja `in_vector`, $v[0], \ldots, v[\texttt{length} - 1]$ pa elemente vektorja `in_out_vector`. Elementi $v[i]$ morajo biti izračunani na naslednji način: $v[i] = u[i] * v[i]$ za vsak $i <$ `length`.

```
int MPI_Op_free (
    MPI_Op *op  /* IN - kazalec na operator */
)
```

Funkcija sprosti pomnilnik in kazalec na operator nastavi na `op = MPI_OP_NULL`.

```
int MPI_Pack (
    void *in_buffer,    /* IN - osnovno sporocilo */
    int elements,       /* IN - stevilo elementov */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    void *out_buffer,   /* IN - zlozeno sporocilo */
    int out_size,       /* OUT - dolzina zlozenega sporocila
                               v bajtih */
    int *offset,        /* IN/OUT - indeks v out_buffer, kjer se
                                  zlaganje zacne */
    MPI_Comm comm       /* IN - komunikator */
)
```

S pomočjo te funkcije lahko nezvezen blok podatkov stisnemo v zvezno strukturo. Potem, ko je sporočilo sprejeto, ga je potrebno še razložiti. Druga možnost je uporaba izpeljanih podatkovnih tipov.

```
int MPI_Pack_size (
    int cnt,            /* IN - stevilo elementov v sporocilu */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    MPI_Comm comm,      /* IN - komunikator */
    int *bound          /* OUT - zgornja meja za velikost zlozenega
                               sporocila */
)
```

Funkcija izračuna zgornjo mejo za velikost zloženega sporočila v bajtih. S klicem te funkcije pred `MPI_Pack` lahko določimo velikost `out_buffer` v slednji.

```
int MPI_Probe (
    int src,           /* rang posiljatelja */
    int tag,           /* IN - oznaka sporocila */
    MPI_Comm comm,     /* IN - komunikator */
    MPI_Status *status /* OUT - status */
)
```

Funkcija preverja, če je na voljo novo sporočilo, vendar ga ne sprejme. Namesto točno določenega pošiljatelja lahko uporabimo `MPI_ANY_SOURCE`, namesto točno določene oznake pa `MPI_ANY_TAG`.

```
int MPI_Recv (
    void *buffer,       /* IN - kazalec na sporocilo */
    int cnt,            /* IN - stevilo elementov v sporocilu */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    int src,            /* IN - rang vira */
    int tag,            /* IN - oznaka sporocila */
    MPI_Comm comm,      /* IN - komunikator */
    MPI_Status *status  /* OUT - status */
)
```

Funkcija je namenjena sprejemanju podatkov. Izvajanje funkcije se ne zaključi dokler podatkov ne sprejme.

```
int MPI_Recv_init (
    void *buffer,       /* IN - kazalec na sporocilo */
    int cnt,            /* IN - stevilo elementov v sporocilu */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    int src,            /* IN - rang vira */
    int tag,            /* IN - oznaka sporocila */
    MPI_Comm comm,      /* IN - komunikator */
    MPI_Request *handle /* OUT - kazalec na zahtevo */
)
```

Funkcija ustvari stalno zahtevo za sprejemanje podatkov. Funkcija je uporabna, kadar program neprestano kliče funkcijo za sprejem podatkov z istimi argumenti. Sprejemanje se začne s klicem funkcije `MPI_Start`.

```
int MPI_Reduce (
    void *send_buffer,  /* IN - kazalec na oddano sporocilo */
    void *recv_buffer,  /* OUT - kazalec na sprejeto sporocilo
```

```
                                dobi ga samo proces root */
    int cnt,            /* IN - stevilo elementov v sporocilu */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    MPI_Op op,          /* IN - operator */
    int root            /* IN - rang glavnega procesa */
    MPI_Comm comm,      /* IN - komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Njena naloga je, da izvede `cnt` operacij redukcije in rezultat shrani v `recv_buffer` na procesu z rangom `root`.

```
int MPI_Reduce_scatter (
    void *send_buffer,  /* IN - kazalec na oddano sporocilo */
    void *recv_buffer,  /* OUT - kazalec na sprejeto sporocilo
                           dobi ga samo proces root */
    int *recv_cnts,     /* IN - polje s stevilom elementov v
                           sporocilu, ki jih root poslje
                           vsakemu procesu */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    MPI_Op op,          /* IN - operator */
    int root            /* IN - rang glavnega procesa */
    MPI_Comm comm,      /* IN - komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Po izvedeni operaciji redukcije rezultate razprši med procese.

```
int MPI_Request_free (
    MPI_Request *handle /* IN - kazalec na zahtevo */
)
```

Funkcija sprosti pomnilnik, ki ga je zasedala struktura zahteva. Uporablja se v kombinaciji s funkcijami za neprestano komunikacijo.

```
int MPI_Rsend (
    void *buffer,       /* IN - sporocilo */
    int cnt,            /* IN - stevilo elementov */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    int dest,           /* IN - rang ciljnega procesa */
    int tag,            /* IN - oznaka sporocila */
    MPI_Comm comm       /* IN - komunikator */
)
```

Funkcija je namenjena pošiljanju podatkov. Pošiljanje se začne šele potem, ko je ustrezna zahteva za sprejem podatkov že pripravljena (*ang.* ready).

```
int MPI_Rsend_init (
    void *buffer,       /* IN - sporocilo */
    int cnt,            /* IN - stevilo elementov */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    int dest,           /* IN - rang ciljnega procesa */
    int tag,            /* IN - oznaka sporocila */
    MPI_Comm comm,      /* IN - komunikator */
    MPI_Request *handle /* OUT - kazalec na zahtevo */
)
```

Funkcija ustvari stalno zahtevo za sprejemanje podatkov (način ready). Funkcija je uporabna, kadar program neprestano kliče funkcijo za sprejem podatkov z istimi argumenti.

```
int MPI_Scan (
    void *send_buffer,  /* IN - oddano sporocilo */
    void *recv_buffer,  /* OUT - sprejeto sporocilo */
    int cnt,            /* IN - stevilo elementov */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    MPI_Op op,          /* IN - operator */
    MPI_Comm comm       /* IN - komunikator */
)
```

Funkcija v strukturo `recv_buffer` vrne rezultat operacije `op` za elemente sporočila `send_buffer` tistih procesov, ki imajo rang manjši ali enak rangu trenutnega procesa.

```
int MPI_Scatter (
    void *send_buffer,       /* IN - oddano sporocilo */
    int send_cnt,            /* IN - stevilo elementov v oddanem
                                     sporocilu */
    MPI_Datatype send_dtype, /* IN - podatkovni tip */
    void *recv_buffer,       /* OUT - sprejeto sporocilo */
    MPI_Datatype recv_dtype, /* IN - podatkovni tip sprejetega
                                     sporocila */
    int root,                /* IN - rang posiljatelja */
    MPI_Comm comm            /* IN - komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Njena naloga je, da raztrosi strukture med procese tako, da vsak proces dobi enako velik kos. V primeru, da to ni mogoče, je potrebno uporabiti funkcijo `MPI_Scatterv`.

```
int MPI_Scatterv (
```

```
    void *send_buffer,       /* IN - oddano sporocilo */
    int *send_cnts,          /* IN - polje z velikostmi blokov, ki
                                     jih posiljamo procesom */
    int *send_disp,          /* IN - odmiki posameznih blokov od
                                     zacetka strukture send_buffer */
    MPI_Datatype send_dtype, /* IN - podatkovni tip */
    void *recv_buffer,       /* OUT - sprejeto sporocilo */
    MPI_Datatype recv_dtype, /* IN - podatkovni tip sprejetega
                                     sporocila */
    int root,                /* IN - rang posiljatelja */
    MPI_Comm comm            /* IN - komunikator */
)
```

Funkcija spada v skupino funkcij za kolektivno komunikacijo. Njena naloga je, da raztrosi strukture med procese. Število elementov, ki jih dobi proces `j`, je `send_cnts[j]`, položaj prvega elementa, ki ga dobi proces `j`, pa `send_disp[j]`. V primeru, da vsi procesi dobijo enako število elementov, je bolje uporabiti funkcijo `MPI_Scatter`.

```
int MPI_Send (
    void *buffer,       /* IN - sporocilo */
    int cnt,            /* IN - stevilo elementov v sporocilu */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    int dest,           /* IN - rang ciljnega procesa */
    int tag,            /* IN - oznaka sporocila */
    MPI_Comm comm       /* IN - komunikator */
)
```

Funkcija pošilja sporočilo drugemu procesu. Okolje MPI se odloči, ali bo sporočilo preneslo v vmesni pomnilnik ali ga bo direktno poslalo drugemu procesu. V prvem primeru se funkcija konča, ko je sporočilo preneseno v vmesni pomnilnik, v drugem pa šele, ko sporočilo sprejeme drugi proces.

```
int MPI_Send_init (
    void *buffer,       /* IN - sporocilo */
    int cnt,            /* IN - stevilo elementov v sporocilu */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    int dest,           /* IN - rang ciljnega procesa */
    int tag,            /* IN - oznaka sporocila */
    MPI_Comm comm,      /* IN - komunikator */
    MPI_Request *handle /* OUT - kazalec na zahtevo */
)
```

Funkcija ustvari stalno zahtevo za pošiljanje podatkov. Uporabna je, kadar program neprestano izvaja pošiljanje z enakimi parametri. Pošiljanje sporočila sproži klic funkcije `MPI_Start`.

```
int MPI_Sendrecv (
    void *send_buffer,       /* IN - sporocilo za posiljanje */
    int send_cnt,            /* IN - stevilo elementov v oddanem
                                     sporocilu */
    MPI_Datatype send_dtype, /* IN - oddani podatkovni tip */
    int dest,                /* IN - rang procesa, ki mu posiljamo
                                     sporocilo */
    int send_tag,            /* IN - oznaka poslanega sporocila */
    void *recv_buffer,       /* OUT - kazalec na sprejeto
                                      sporocilo */
    int recv_cnt,            /* IN - stevilo elementov v sprejetem
                                     sporocilu */
    MPI_Datatype recv_dtype, /* IN - sprejeti podatkovni tip */
    int src,                 /* IN - rang procesa od katerega
                                     aktivni proces sprejema
                                     sporocilo */
    int recv_tag,            /* IN - oznaka sprejetega sporocila */
    MPI_Comm comm            /* IN - komunikator */
    MPI_Status *status       /* OUT - status prejetega sporocila */
)
```

Funkcija je kombinacija funkcije `MPI_Send` in funkcije `MPI_Recv`. Posebej uporabna je, kadar so procesi organizirani v obroč in vsak proces pošilja sporočilo svojemu sosedu. Z uporabo kombinirane funkcije se elegantno izognemo morebitnemu smrtnemu objemu. Kadar želimo, da sta naslova oddanega in sprejetega sporočila enaka, je bolje uporabiti funkcije `MPI_Sendrecv_replace`.

```
int MPI_Sendrecv_replace (
    void *buffer,       /* IN/OUT - naslov sporocila  */
    int cnt,            /* IN - stevilo elementov v oddanem
                              sporocilu */
    MPI_Datatype dtype, /* IN - oddani podatkovni tip */
    int dest,           /* IN - rang procesa, ki mu posiljamo
                              sporocilo */
    int send_tag,       /* IN - oznaka poslanega sporocila */
    int src,            /* IN - rang procesa od katerega aktivni
                              proces sprejema sporocilo */
    int recv_tag,       /* IN - oznaka sprejetega sporocila */
    MPI_Comm comm       /* IN - komunikator */
    MPI_Status *status  /* OUT - status prejetega sporocila */
)
```

Funkcija izvaja kombinacijo operacij pošiljanja in sprejemanja. Podobna je funkciji `MPI_Sendrecv`, za razliko od nje pa sprejeto sporočilo zamenja staro sporočilo na naslovu `buffer`. Dolžina oddanega in sprejetega sporočila mora biti seveda enaka.

```
int MPI_SSend (
    void *buffer,       /* IN - sporocilo */
    int cnt,            /* IN - stevilo elementov v sporocilu */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    int dest,           /* IN - rang ciljnega procesa */
    int tag,            /* IN - oznaka sporocila */
    MPI_Comm comm       /* IN - komunikator */
)
```

Funkcija pošilja sporočilo drugemu procesu v sinhronem načinu. To pomeni, da se izvaja toliko časa, dokler proces, ki mora sprejeti sporočilo, ne začne postopka sprejemanja le-tega.

```
int MPI_SSend_init (
    void *buffer,       /* IN - sporocilo */
    int cnt,            /* IN - stevilo elementov v sporocilu */
    MPI_Datatype dtype, /* IN - podatkovni tip */
    int dest,           /* IN - rang ciljnega procesa */
    int tag,            /* IN - oznaka sporocila */
    MPI_Comm comm,      /* IN - komunikator */
    MPI_Request *handle /* OUT - kazalec na zahtevo */
)
```

Funkcija ustvari stalno zahtevo za pošiljanje podatkov v sinhronem načinu. Uporabna je, kadar program neprestano izvaja pošiljanje z enakimi parametri. Pošiljanje sporočila sproži klic funkcije `MPI_Start`.

```
int MPI_Start (
    MPI_Request *handle /* IN - kazalec na zahtevo */
)
```

Funkcija sproži stalno komunikacijsko zahtevo, ki jo podamo kot kazalec na strukturo zahteve tipa `MPI_Request`.

```
int MPI_StartAll (
    int size,            /* IN - stevilo zahtev */
    MPI_Request *handles /* IN - polje kazalcev na zahteve */
)
```

Funkcija sproži stalne komunikacijske zahteve. Za razliko od funkcije `MPI_Start` lahko sproži več zahtev hkrati.

```
int MPI_Test (
    MPI_Request *handle, /* IN - kazalec na zahtevo */
    int *flag,           /* OUT - podatek o zakljucitvi zahteve */
    MPI_Status *status   /* OUT - rezultat prenosa */
)
```

Funkcija preveri, če se je komunikacijska operacija, povezana z zahtevo `handle`, zaključila. V primeru, da se je zaključila, lahko s pregledom strukture `status` ugotovimo pošiljatelja, oznako sporočila in kodo napake.

```
int MPI_Test_cancelled (
    MPI_Request *handle, /* IN - kazalec na zahtevo */
    int *flag,           /* OUT - podatek o zakljucitvi zahteve */
)
```

Funkcija preko parametra `flag` sporoči, če so bile vse komunikacijske zahteve uspešno prekinjene.

```
int MPI_Test_all (
    int size,            /* IN - stevilo zahtev */
    MPI_Request *handle, /* IN - polje kazalcev na zahtevo */
    int *flag,           /* OUT - podatek o zakljucitvi zahteve */
    MPI_Status *status   /* OUT - polje rezultatov prenosa */
)
```

Funkcija preveri, če so se vse komunikacijske operacije, povezane s `size` zahtevami, navedenimi v polju `handle`, zaključile.

```
int MPI_Test_any (
    int size,            /* IN - stevilo zahtev */
    MPI_Request *handle, /* IN - polje kazalcev na zahteve */
    int *index,          /* OUT - indeks zakljucene zahteve */
    int *flag,           /* OUT - podatek o zakljucitvi zahteve */
    MPI_Status *status   /* OUT - rezultat prenosa */
)
```

Funkcija preveri, če se je vsaj ena komunikacijska zahteva zaključila. V spremenljivki `index` vrne indeks zaključene zahteve. V primeru, da je zaključena operacija sprejema, lahko dodatne podrobmosti o sporočilu dobimo iz spremenljivke `status`.

```
int MPI_Test_some (
    int in_cnt,          /* IN - stevilo zahtev */
    MPI_Request *handle, /* IN - polje kazalcev na zahteve */
    int *out_cnt,        /* OUT - stevilo koncanih zahtev */
    int *index,          /* OUT - polje indeksov zakljucenih
                                  zahtev */
    MPI_Status *status   /* OUT - polje rezultatov prenosa */
)
```

Funkcija preko parametra `index` sporoča, katere komunikacijske zahteve so se že zaključile.

```
int MPI_Topo_test (
    MPI_Comm comm, /*IN - komunikator */
    int *topology  /*OUT - tip komunikatorja */
)
```

Funkcija vrne tip komunikatorja, ki je lahko `MPI_GRAPH`, `MPI_CART`, `MPI_UNDEFINED`.

```
int MPI_Type_commit (
    MPI_Datatype dtype /* IN - podatkovni tip */
)
```

Funkcija pripravi izpeljani podatkovni tip tako, da je primeren za prenašanje s sporočili.

```
int MPI_Type_contiguous (
    int cnt,          /* IN - stevilo kopij */
    MPI_Datatype old, /* IN - stari podatkovni tip */
    MPI_Datatype *new /* OUT - nov podatkovni tip */
)
```

Funkcija sestavi nov podatkovni tip kot več kopij starega podatkovnega tipa.

```
int MPI_Type_count (
    MPI_Datatype dtype, /* IN - stari podatkovni tip */
    int *cnt            /* OUT - stevilo podatkovnih tipov */
)
```

Funkcija vrne število podatkovnih tipov, ki na najvišjem nivoju sestavljajo podatkovni tip `dtype`.

```
int MPI_Type_extent (
    MPI_Datatype dtype, /* IN - stari podatkovni tip */
    MPI_Aint *extent    /* OUT - dolzina v bajtih */
)
```

Funkcija vrne dolžino podatkovnega tipa v bajtih, zaokroženo navzgor tako, da ustreza poravnavanju podatkov na uporabljeni strojni opremi.

```
int MPI_Type_free (
    MPI_Datatype dtype /* IN - stari podatkovni tip */
)
```

Funkcija sprosti pomnilnik, ki je bil dodeljen podatkovnemu tipu `dtype`, in postavi `dtype` na vrednost `MPI_DATATYPE_NULL`. Vse komunikacijske zahteve, ki uporabljajo podatkovni tip in sledijo funkciji, se bodo zaključile normalno.

```
int MPI_Type_hindexed (
    int cnt,              /* IN - stevilo blokov */
    int *block_len_array, /* IN - polje s stevilom elementov po
                                  blokih */
    MPI_Aint *disp_array, /* IN - polje odmikov blokov v bajtih */
    MPI_Datatype old,     /* IN - star podatkovni tip */
    MPI_Datatype *new     /* OUT - nov podatkovni tip */
)
```

Funkcija je enaka funkciji `MPI_Type_indexed`, le da so odmiki podani v bajtih.

```
int MPI_Type_hvector (
    int cnt,              /* stevilo blokov */
    int len,              /* stevilo elementov na blok */
    MPI_Aint *step,       /* razlika odmikov dveh zacetkov blokov */
    MPI_Datatype old,     /* star podatkovni tip */
    MPI_Datatype *new     /* nov podatkovni tip */
)
```

Funkcija zgradi nov podatkovni tip z repliciranjem starega podatkovnega tipa. Nov podatkovni tip je sestavljen iz `cnt` blokov, od katerih ima vsak `len` kopij starega podatkovnega tipa. `step` podaja razmik v bajtih med dvema blokoma.

```
int MPI_Type_indexed (
    int cnt,              /* IN - stevilo blokov */
    int *block_len_array, /* IN - polje s stevilom elementov po
                                  blokih */
    int *disp_array,      /* IN - polje odmikov blokov v
                                  dolzinah old */
    MPI_Datatype old,     /* IN - star podatkovni tip */
    MPI_Datatype *new     /* OUT - nov podatkovni tip */
)
```

Funkcija sestavi nov podatkovni tip kot zaporedje enega ali več blokov, od katerih je vsak sestavljen iz enega ali več kopij starega podatkovnega tipa. `cnt` določa, iz koliko blokov je sestavljen novi podatkovni tip, polje `block_len_array` pa določa število podatkovnih tipov na blok. Odmiki v polju odmikov se podajajo v dolžinah podatkovnega tipa `old`.

```
int MPI_Type_lb (
    MPI_Datatype dtype, /* IN - podatkovni tip */
    MPI_Aint *lb        /* OUT - spodnji odmik */
)
```

Funkcija vrne odmik od izhodišča za spodnjo mejo podatkovnega tipa `dtype`.

```
int MPI_Type_size (
    MPI_Datatype dtype, /* IN - podatkovni tip */
    int *size           /* OUT - skupna velikost */
)
```

Funkcija vrne skupno velikost podatkovnega tipa v bajtih.

```
int MPI_Type_struct (
    int cnt,             /* IN - stevilo blokov v new */
    int *block_len,      /* IN - polje s stevilom elementov
                                 na blok */
    MPI_Aint *disp,      /* IN - odmik za vsak blok */
    MPI_Datatype *dtype, /* IN - polje podatkovnih tipov */
    MPI_Datatype *new    /* OUT - nov podatkovni tip */
)
```

Funkcija zgradi nov podatkovni tip, ki je sestavljen iz `cnt` blokov podatkovnih tipov. Vsak blok vsebuje `block_len[j]` podatkovnih tipov `dtype[j]`.

```
int MPI_Type_ub (
    MPI_Datatype dtype, /* IN - podatkovni tip */
    MPI_Aint *ub        /* OUT - zgornji odmik */
)
```

Funkcija vrne odmik od izhodišča za zgornjo mejo podatkovnega tipa `dtype`.

```
int MPI_Type_vector (
    int cnt,          /* IN - stevilo blokov v new */
    int block_len,    /* IN - polje s stevilom elementov na blok */
    int step,         /* IN - stevilo elementov med zacetkoma dveh
                              blokov */
    MPI_Datatype old, /* IN - stari podatkovni tip */
    MPI_Datatype *new /* OUT - nov podatkovni tip */
)
```

Funkcija zgradi nov podatkovni tip z replikacijo starega podatkovnega tipa. Nov podatkovni tip ima `cnt` blokov dolžine `block_len`. Razalja med dvema začetkoma blokov `step` je podana kot večkratnik dolžine podatkovnega tipa `old`.

```
int MPI_Unpack (
    void *in_buffer,    /* IN - vhodni pomnilnik */
    int len,            /* IN - dolzina vhodnega pomnilnika */
    int *position,      /* IN/OUT - polzaj v vhodnem pomnilniku */
    void *out_buffer,   /* OUT - izhodni pomnilnik */
```

```
    int out_cnt,        /* IN - stevilo elementov za razlo\v zitev */
    MPI_Datatype dtype, /* IN podatkovni tip */
    MPI_Comm comm       /* komunikator */
)
```

Funkcija razloži sporočilo iz `in_buffer` v `out_buffer`. Na začetku parameter `position` označuje začetek zloženega sporočila, ko se funkcija zaključi, pa označuje prvi prosti bajt za razloženim sporočilom.

```
int MPI_Wait (
    MPI_Request *handle, /* IN - kazalec na zahtevo */
    MPI_Status *status   /* OUT - stanje zahteve */
)
```

Funkcija zaključi operacije, ki ne zaustavijo nadaljnega izvajanja programa. Če gre za operacijo pošiljanja, funkcija čaka, dokler sporočilo ni odposlano. Če gre za sprejemanje, funkcija čaka, dokler sporočilo ni sprejeto.

```
int MPI_Wait_all (
    int cnt,             /* IN - stevilo obravnavanih zahtev */
    MPI_Request *handle, /* IN - polje kazalcev na zahteve */
    MPI_Status *status   /* OUT - status zakljucenih zahtev */
)
```

Funkcija se izvaja, dokler se ne zaključi vseh `cnt` zahtev, navedenih v polju `handle`. V `status` je shranjena informacija o vseh zahtevah.

```
int MPI_Wait_any (
    int cnt,             /* IN - stevilo obravnavanih zahtev */
    MPI_Request *handle, /* IN - polje kazalcev na zahteve */
    int *index           /* OUT - indeks zakljucene zahteve */
    MPI_Status *status   /* OUT - status zakljucenih zahtev */
)
```

Funkcija se izvaja, dokler se ne konča vsaj ena zahteva. Vrne njen indeks v polju in status.

```
int MPI_Wait_some (
    int cnt,             /* IN - stevilo obravnavanih zahtev */
    MPI_Request *handle, /* IN - polje kazalcev na zahteve */
    int *out_cnt,        /* OUT - stevilo zakljucenih zahtev */
    int *index_array     /* OUT - polje indeksov zakljucenih
                                  zahtev */
    MPI_Status *status   /* OUT - status zakljucenih zahtev */
)
```

Funkcija se izvaja, dokler se ne konča ena ali več zahtev. Za vse končane zahteve vrne indekse in status.

```
double MPI_Wtick ( void )
```

Funkcija vrne natančnost ure v sekundah.

```
double MPI_Wtime ( void )
```

Funkcija vrne uro v sekundah, merjeno od neke točke v preteklosti naprej. Točka v preteklosti se med izvajanjem procesa ne spreminja. Čas, ki ga program potrebuje za izvajanje nekega bloka, določimo kot razliko izmerjenega časa ob končanju izvajanja bloka in izmerjenega časa ob začetku izvajanja bloka.

Dodatek C

Funkcije knjižnice OpenMP

V tem dodatku so opisane vse funkcije jezikov C/C++, ki jih ponuja knjižnica OpenMP. Funkcije imajo največ en parameter. Vsi parametri so vhodni, njihove vrednosti pred klicem funkcije nastavi klicatelj.

`int omp_get_dynamic (void)`
Funkcija vrne 1, če je dinamično kreiranje niti omogočeno, in 0, če ni.

`int omp_get_max_threads (void)`
Funkcija vrne maksimalno število niti, ki jih program med delovanjem lahko ustvari.

`int_omp_get_nested (void)`
Funkcija vrne 1, če je omogočena gnezdena paralelizacija. Nobena trenutna implementacija knjižnice OpenMP gnezdene paralelizacije ne podpira.

`int omp_get_num_procs (void)`
Funkcija vrne število procesorjev, ki jih lahko uporablja paralelni program.

`int omp_get_num_threads (void)`
Funkcija vrne število trenutno aktivnih niti. Če jo kličemo iz zaporednega dela kode, vrne 1.

`int omp_get_thread_num (void)`
Funkcija vrne oznako niti. Če je aktivnih `t` niti, so te označene z indeksi od `0` do `t-1`.

`int omp_in_parallel (void)`
Funkcija vrne 1, če je klicana iz paralelnega bloka, drugače vrne 0.

`void omp_set_dynamic (int k)`
S to funkcijo omogočimo (`k = 1`) ali onemogočimo (`k = 0`) dinamično kreiranje niti.

Če je dinamično ustvarjanje niti omogočeno, je med izvajanjem programa običajno aktivnih toliko niti, kolikor je v sistemu procesorjev. Če želimo sami nadzorovati število aktivnih niti, moramo dinamično ustvarjanje niti onemogočiti.

`void omp_set_nested (int k)`
S to funkcijo omogočimo (`k = 1`) ali onemogočimo (`k = 0`) gnezdeno paralelizacijo. Vse trenutne implementacije knjižnice OpenMP podpirajo samo en nivo paralelizacije. Nobena trenutna implementacija knjižnice OpenMP gnezdene paralelizacije ne podpira, zato omogočenje gnezdene paralelizacije nima nobenega vpliva na izvajanje programa.

`void omp_set_num_threads (int t)`
S to funkcijo nastavimo število niti, ki naj jih program uporablja v nadaljnjih paralelnih blokih. Število niti lahko preseže število procesorjev. V tem primeru se več niti izvaja na enem procesorju. Število niti moramo nastaviti v zaporednem delu programske kode.

Literatura

[1] J. von Neumann: Theory of Self-Reproducing Automata, University of Illinois Press, Illinois, 1966.

[2] S. Wolfram: Statistical mechanics of cellular automata, Reviews of Modern Physics, vol. 55, no. 3, 601-644, 1983.

[3] S. Wolfram: Cellular automata as models of complexity, Nature, vol. 311, 419-424, 1984.

[4] S. Wolfram: A new kind of science, Wolfram Media, 2002.

[5] A. Dobnikar, S. Vavpotič, A. Likar: Dynamic systems modeling with stochastic cellular automata - evolutionary versus stochastic correlation approach, CIT J. Comput. Inf. Technol., vol. 10, 251-259, 2002.

[6] S. Wolfram: Theory and Applications of Cellular Automata: World Scientific, 1986.

[7] I. Jeras, A. Dobnikar: Algorithms for computing preimages of cellular automata configurations, Physica D, vol. 233, no. 2, 95-111, 2007.

[8] J. H. Holland: Adaptation in Natural and Artificial Systems, The University of Michigan press, Michigan, 1975.

[9] E. F. Codd: Cellular Automata, Academic Press, 1968.

[10] M. Sipper: Evolution of Parallel Cellular Machines - The Cellular Programming Approach, Springer, 1997.

[11] M. L. Minsky: Computation: Finite and Infinite Machines, Prentice-Hall, Englewood Cliffs, New Jersey, 1967.

[12] E. R. Banks: Universality in cellular automata, 11th IEEE Ann. Symp. On Switching and Automata, Santa Nonica, California, 194-215, 1970.

[13] F. C. Richards, T. P. Meyer, N. H. Packard: Extracting cellular automaton rules directly from experimental data, Physica D, vol. 45, 189-202, 1990.

[14] P. Gacs, G. L. Kurdyumov, L. A. Levin: One-dimensional uniform array that wash out finite islands, Problemy Peredachi Informatsii, vol. 14, 92-98, 1978.

[15] S. A. Kauffman: Metabolic stability and epigenesis in randomly constructed genetic nets, Journal of Theoretical Biology, vol. 22, 437-467, 1969.

[16] M. J. Quinn: Parallel Programming in C with MPI and OpenMP, McGraw Hill, 2003.

[17] M. J. Flynn, K. W. Rudd: Parallel architectures, ACM Computing Surveys 28(1), 67-70, 1996.

[18] I. Foster: Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering, Addison-Wesley, 1995.

[19] P. S. Pacheco: Parallel Programming with MPI, Morgan Kaufmann Publishers, 1997.

[20] W. Gropp, E. Lusk, A. Skjellum: Using MPI, MIT Press, 1999.

[21] OpenMP forum: http://www.openmp.org.

[22] S. Bornholdt, H. G. Schuster (ur.): Handbook of Graphs and Networks, Viley, 2003.

[23] S. N. Dorogovtsev, J. F. F. Mendes: Evolution of Networks, Advances in Physics, 51 (4) 1079-1187, 2002.

[24] D. J. Watts, Small Worlds, Princeton University, 2004.

[25] R. Steinmetz, K. Wehrle (ur.): Peer-to-Peer Systems and Applications, Springer, 2005.

[26] G. Coulouris, J. Dollimore, T. Kindberg: Distributed Systems-Concepts and Design, Addison-Wesley, 2005.

[27] M. Weiser: The Computer for the 21st Century, Scientific America, vol. 265, no. 3, 94-104, 1991.

[28] T. R. Halfhill: Parallel Processing with CUDA, Microprocessor Report, Jan. 28, 2008.

Stvarno kazalo